新手学编程ABC丛书

C#编程新手自学手册

陈强 等编著

附赠DVD ROM光盘
- 全书PPT教学资源
- 610分钟讲解视频
- 300多个源代码文件
- 90多个实例拓展

进入职场 就是现在！


机械工业出版社
CHINA MACHINE PRESS

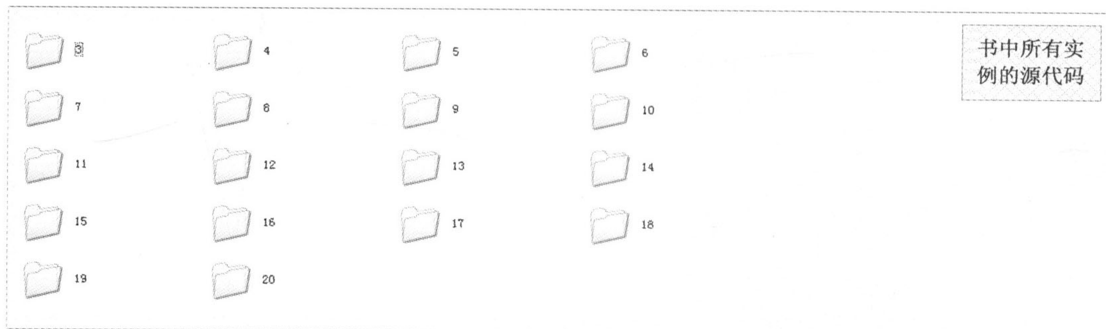


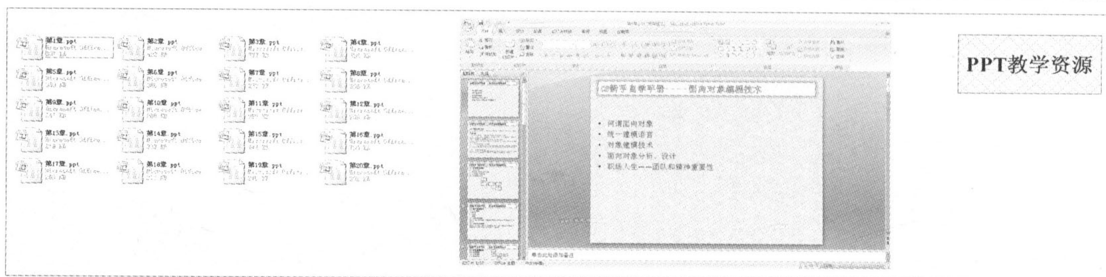
VISIT…

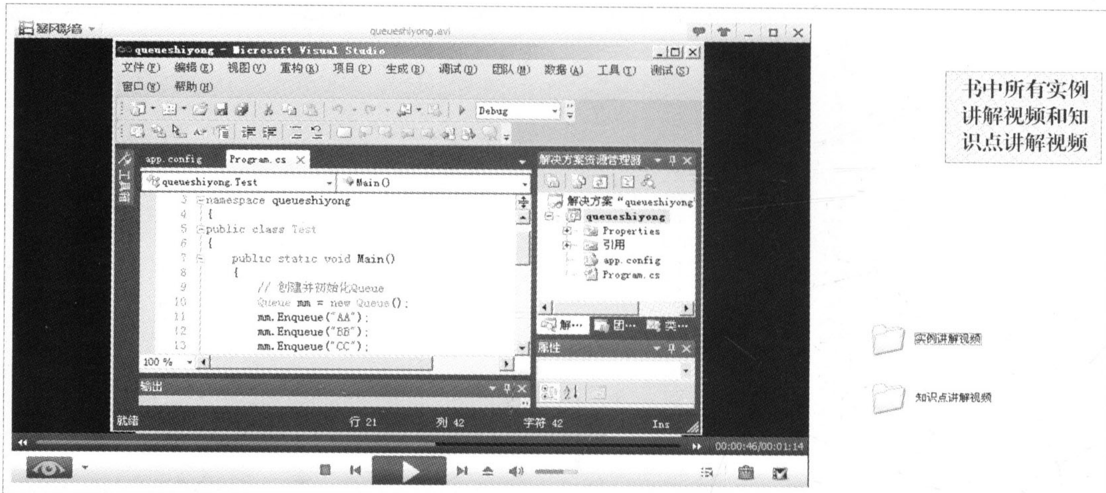
LANZAROTE Caliente.com

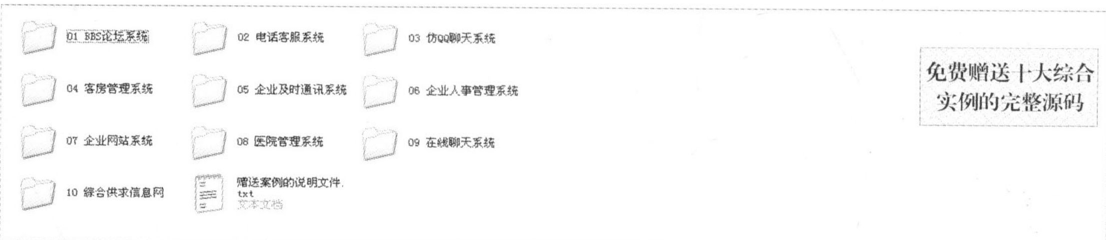
随书光盘内容说明

在本书 DVD 光盘中保存了本书所需的教学资源，具体如下图。





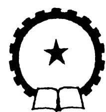




新手学编程 ABC 丛书

C#编程新手自学手册

陈　强　等编著



机 械 工 业 出 版 社

C#是当今使用最为频繁的编程语言之一，一直在开发领域占据重要的地位。本书循序渐进、由浅入深地详细讲解了 C#的核心技术，并通过具体实例的实现过程演示了各个知识点的具体使用流程。全书共 20 章，分为 4 篇。第 1～6 章是基础篇，介绍了 C# 开发所必须具备的基本知识，逐一讲解了 C# 开发基础、C# 开发环境、变量与常量、表达式与运算符、流程控制语句、面向对象编程技术等知识；第 7～11 章是核心技术篇，逐一讲解了方法、类、集合和数组应用、继承与多态、字符串与正则表达式等知识；第 12～19 章是提高篇，逐一讲解了委托与事件、泛型、文件操作、Windows 窗体编程、Web 应用编程、数据库和 ADO.NET 操作、DataGrid 和数据绑定、GDI+图形图像编程等知识；第 20 章是综合实战篇，通过一个综合实例——俄罗斯方块游戏的实现过程，详细讲解了 C#语言在日常项目开发中的综合应用流程，并穿插介绍了项目的实现技巧。全书采用知识性与趣味性相结合的对话讲解方式，穿插了 C#学习技巧和职场生存法则，引领读者全面掌握 C#。

本书不但适用于 C#的初学者，也适用于有一定 C#基础的读者，同时可以作为程序员的参考用书。

随书附赠的 DVD 光盘配有 PPT 电子课件、教学视频及程序源代码。

图书在版编目（CIP）数据

C#编程新手自学手册 / 陈强等编著．—北京：机械工业出版社，2012.1

（新手学编程 ABC 丛书）

ISBN 978-7-111-36370-5

Ⅰ．①C…　Ⅱ．①陈…　Ⅲ．①C 语言－程序设计－手册　Ⅳ．①TP312-62

中国版本图书馆 CIP 数据核字（2011）第 228327 号

机械工业出版社（北京市百万庄大街 22 号　邮政编码 100037）

策划编辑：丁　诚　杨　硕

责任编辑：丁　诚　杨　硕

责任印制：杨　曦

保定市中画美凯印刷有限公司印刷

2012 年 1 月第 1 版 • 第 1 次印刷

184mm×260mm • 37 印张 • 916 千字

0001－3500 册

标准书号：ISBN 978-7-111-36370-5

ISBN 978-7-89433-268-4（光盘）

定价：86.00 元（含 1DVD）

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

电话服务

社服务中心：（010）88361066

销售一部：（010）68326294

销售二部：（010）88379649

读者购书热线：（010）88379203

网络服务

门户网：http://www.cmpbook.com

教材网：http://www.cmpedu.com

封面无防伪标均为盗版

从 书 序

从《杜拉拉升职记》谈起

近年来，职场小说备受青睐，李可老师的《杜拉拉升职记》更是受到广大读者的喜爱，还被搬上了银幕。对许多人来说，职场生涯占据了整个人生的很大一部分时间，怎样才能在职场中如鱼得水，是人们必须认真思考的重要问题。即将走上程序员岗位的读者朋友们，请自问：是否已经对未来的职场生涯胸有成竹？

本丛书不但可以帮助初学者提前演练职场生活，而且对在职人员也有借鉴意义。技术方面的知识就不用再多说了，每一页所包含的内容都是作者多年来的技术结晶。希望读者在阅读本丛书后，不但能学到编程技术，而且能够提前体验到职场中的一些常见场景，为将来的职场生涯做一些准备。希望本书能为读者们解惑，也希望能激励读者们在这个行业继续奋斗下去，迎接大家的将是明媚的阳光。

程序员的各个阶段

按照掌握技术的熟练程度来划分程序员的不同阶段，可以大体分为 5 个阶段。

1）初学者：处在此阶段的可能是一名在校学生，可能是应届毕业生，也可能是准备从其他行业向编程行业转行的人员。其共同特点是刚开始学习编程知识，对每一个知识点都充满了好奇，对未来充满期望。

2）菜鸟：这里的菜鸟可能是技术菜鸟，也可能是职场菜鸟。特点是某项技术的基本知识已经学习完毕，但是没有经过项目的洗礼，尚需要实战演练来磨练。这个阶段一般指处于试用期或者刚刚从事程序员工作的人员。

3）初级程序员：对项目开发的基本流程有了初步的认识，并通过工作实战演练了自己的技术。此阶段处于进一步与同事、上级、下级和客户交流的摸索阶段，也是逐渐融入职场的阶段。

4）高级程序员：开发经验丰富，技术扎实，对同事关系、上下级关系和客户关系已经如鱼得水，也是事业发展的瓶颈阶段。此阶段的程序员在职场中一般是软件高级工程师。

5）资深程序员：技术实力和人脉关系俱佳，一个项目任务能如探囊取物般轻松完成。但是也对自己的未来充满思索，想寻求待遇更好的职位，会考虑跳槽，也会考虑创业。为了表述得更加直观，下面通过一幅图来展示程序员的成长历程。

本丛书书目

根据综合考虑分析，本丛书首批书目如下。

C 语言编程新手自学手册

C#编程新手自学手册

Visual C++编程新手自学手册

Java Web 编程新手自学手册

Java 编程新手自学手册

PHP 编程新手自学手册

编程算法新手自学手册

技术菜鸟或职场菜鸟。特点是某项技术的基本知识已经学习完毕，但是没有经过项目的洗礼，尚需要实战演练来磨练。



处在此阶段的可能是一名在校学生，可能是应届毕业生，也可能是准备从其他行业向编程行业转行的人员。其共同特点是刚开始学习编程知识，对每一个知识点都充满了好奇，对未来充满期望。



对项目开发的基本流程有了自己的认识，并通过工作实战演练了初步的技术。此阶段处于进一步与同事、上级、下级和客户交流的摸索阶段，也是逐渐融入职场的阶段。



开发经验丰富，技术扎实，对同事关系、上下级关系和客户关系已经如鱼得水，但也是事业进一步发展的瓶颈阶段。

技术实力和人脉关系俱佳，一个项目任务能如探囊取物般轻松完成。对自己的未来充满思索，想寻求待遇更好的职位。会考虑跳槽，也会考虑创业。



致读者

学习程序开发之路是充满挑战之路，也是充满乐趣之路，这条路没有捷径可走。梦想像《天龙八部》中虚竹那样轻松获得一甲子功力，是不现实的。读者们要想真正学好编程，需要付出辛勤的汗水。根据笔者的亲身体会，替读者总结出 3 条学习编程的建议。

（1）培养兴趣

无论做什么事情，只要有了兴趣，就喜欢花费时间去做它。只要喜欢感受那调试成功的喜悦，就说明已经对编程产生了兴趣。这种喜悦会使自己更加喜欢编程，会带来成就感。闲暇时刻建议多去专业编程论坛逛一逛，灌灌水。论坛里的朋友们不但能帮助解决问题，而且还能带来其他非技术性的收获。

（2）脚踏实地

欲速则不达，学编程切忌有浮躁的心态。很多初学者刚学会了基本语法知识，调试成功了几段代码，就迫不及待地大声宣布："我精通××语言了"。但是当遇到问题之后才发现，自己学到的只是九牛一毛。常说"书山有路勤为径，学海无涯苦作舟"，是很有道理的。

（3）多实践

程序开发很强调实际动手能力，所以实践就变得尤为重要。有前辈高人认为，学习编程的秘诀是"编程、编程、再编程，练习，练习，再练习"，笔者深表赞同。学编程不仅要多实践，而且要快实践。在看书的时候，不要等到完全理解了才动手，而是应该在看书的同时敲代码，程序运行的各种情况可以让自己更快、更牢固地掌握知识点。

我们的服务邮箱是 150649826@qq.com，读者在阅读本丛书时，如果发现错误或遇到问题，可以发送电子邮件及时与我们联系，我们会尽快给予答复。

丛书编委会

前　言

C#语言的重要性

微软 C#语言的定义主要是从 C 和 C++继承而来的，而且语言中的许多元素也反映了这一点。C#在设计者从 C++继承的可选选项方面比 Java 要广泛一些，它还加入了自己独有的新特点。

最近微软公司推出了 Visual Studio 2010。Visual Studio 2010 是微软公司精心打造的新一代软件开发平台产品。该平台的核心开发语言是 Visual C# 2010，又称做 C# 4.0。Visual Studio 2010 不仅继承了 Visual Studio 以前版本的众多优点，而且引入了多个新增特性。这些新特性使开发平台的环境更加人性化，并且在编写、调试程序方面的功能也大大增强。

本书的特色

本书的最大特色是以作者学习历程的日记为主线，用一问一答的模式引出知识点。本书讲解了作者从一个即将毕业的大学生学习编程开始，到顺利毕业并进入职场的成长历程。作者用日记的形式全程记录了过去的学习历程。日记和学习、职场密切相关，也和知识点相关，使读者读后有豁然开朗的感觉。

（1）以成长经历为主线

本书以日记的方式记录下了学习过程中的点点滴滴，趣味性和知识性并重，情节引人入胜。

（2）理论加实践的学习模式

书中遵循了理论加实践的写作模式，在每个知识点讲解完毕后，都会用一个具体实例来演示知识点的用法。所有实例都具有典型性和代表性。

（3）传授学习和职场经验

书中介绍了一些职场中的规则，逐一向读者传授了学习，应聘，处理同事、客户、上下级关系，跳槽，创业和升职的经验和体会，可给读者以启示。

（4）给读者以最大实惠

本书配套光盘中不但有书中实例的源代码，而且有全程视频讲解的 PPT 素材，并为书中的项目案例都配备了详细的视频讲解。

本书的内容

第 1～6 章是基础篇，介绍 C#开发所必须具备的基本知识，逐一讲解了 C#开发基础，C#开发环境，变量与常量，表达式与运算符，流程控制等知识。

第 7～11 章是核心技术篇，逐一讲解了方法，类，集合和数组应用，继承与多态，字符串与正则表达式等知识。

第 12～19 章是提高篇，逐一讲解了委托与事件，泛型，文件操作，Windows 窗体编程，Web 应用编程，数据库和 ADO.NET 操作，DataGrid 和数据绑定，GDI+图形图像编程等知识。

第 20 章是综合实战篇，通过一个综合实例——俄罗斯方块游戏的实现过程，详细讲解了 C#语言在日常项目开发中的综合应用流程。

本书的读者对象

初学编程的自学者
编程爱好者
大中专院校的老师和学生
相关培训机构的老师和学员
毕业设计的学生
初中级程序开发人员
程序测试及维护人员
参加实习的初级程序员
在职程序员
资深程序员

致谢

参与本书编写的人员有张兴建、周秀、王梦、管西京、张子言、朱万林、李强、王孟、陈德春、周涛、刘海洋、关立勋、孟娜、王石磊、徐亮、张储、蒋凯、扶松柏、唐凯、焦甜甜、张斌、杨国华、杨絮、谭贞军、张玲玲、陈强。

在本书的编写过程中，我们始终本着科学、严谨的态度，力求精益求精，但错误、疏漏之处在所难免，敬请广大读者批评指正。我们的服务邮箱是 150649826@qq.com，读者在阅读本书时，如果发现错误或遇到问题，可以发送电子邮件及时与我们联系，我们会尽快给予答复。

感谢您购买本书，希望本书能成为您编程路上的领航员，祝您读书快乐！

编　者

目　　录

丛书序
前言

第一篇　基　础　篇

第 1 章　C#开发基础 1
1.1　什么是 C# 2
1.1.1　C#的推出背景 2
1.1.2　C#的特点 2
1.2　.NET Framework 框架 4
1.2.1　.NET Framework 简介 4
1.2.2　不得不知的几个概念 5
1.3　疑难问题解析 6
职场点拨——C#的地位 7
第 2 章　初识 C#开发环境 9
2.1　Visual Studio 2010 介绍 9
2.1.1　Visual Studio 2010 工具介绍 9
2.1.2　Visual Studio 2010 工具的内容 10
2.1.3　Visual Studio 2010 新特性 11
2.1.4　Visual Studio 2010 与 2008 版本的对比 12
2.1.5　安装 Visual Studio 2010 14
2.2　初步认识 Visual Studio 2010 开发环境 15
2.2.1　Visual Studio 2010 设置 15
2.2.2　新建项目 16
2.2.3　解决方案和类视图 19
2.2.4　文本编辑器 20
2.2.5　生成与查错 22
2.3　疑难问题解析 23
职场点拨——怎样学好 C# 24
第 3 章　变量与常量 26
3.1　C#的基本语法 26
3.2　变量 29
3.2.1　C#类型 29
3.2.2　给变量命名 34
3.2.3　变量的声明和赋值 35
3.3　常量 36

3.4 类型转换 36
3.4.1 隐式转换 36
3.4.2 显式转换 38
3.4.3 装箱与拆箱 40
3.5 复杂的变量类型 42
3.5.1 枚举 42
3.5.2 结构 44
3.5.3 数组 45
3.6 基本.NET 框架类 47
3.6.1 Console 类 47
3.6.2 Convert 类 52
3.6.3 Math 类 53
3.7 疑难问题解析 54
职场点拨——和客户相处之道 55
第 4 章 表达式与运算符 56
4.1 表达式 56
4.2 运算符 57
4.2.1 基本运算符 57
4.2.2 数学运算符 59
4.2.3 赋值运算符 61
4.2.4 比较运算符 61
4.2.5 逻辑运算符 62
4.2.6 移位运算符 63
4.2.7 三元运算符 63
4.2.8 运算符的优先级 65
4.3 疑难问题解析 66
职场点拨——生活方式决定人生道路 66
第 5 章 流程控制语句 68
5.1 语句和语句块 68
5.2 选择语句 69
5.2.1 使用 if 语句 69
5.2.2 使用 switch 语句 70
5.3 循环语句 72
5.3.1 使用 while 语句 72
5.3.2 使用 do…while 语句 74
5.3.3 使用 for 语句 77
5.4 跳转语句 79
5.4.1 使用 break 语句 80
5.4.2 使用 continue 语句 81

5.4.3 使用 return 语句 …… 83
5.4.4 使用 goto 语句 …… 84
5.5 疑难问题解析 …… 85
职场点拨——创业还是就业？ …… 85
第 6 章 面向对象编程技术 …… 87
6.1 何谓面向对象 …… 88
6.1.1 知道 OOP 吗 …… 88
6.1.2 C#面向对象编程 …… 88
6.2 统一建模语言 …… 88
6.2.1 UML 简介 …… 89
6.2.2 类图和对象图 …… 89
6.2.3 序列图和状态图 …… 91
6.3 对象建模技术 …… 92
6.3.1 功能模型 …… 93
6.3.2 对象模型 …… 93
6.3.3 动态模型 …… 93
6.4 面向对象分析 …… 94
6.4.1 面向对象分析的原则 …… 94
6.4.2 面向对象分析的阶段 …… 95
6.4.3 面向对象分析的具体目标 …… 96
6.4.4 面向对象分析的过程和步骤 …… 97
6.5 面向对象设计 …… 97
6.5.1 面向对象设计概述 …… 97
6.5.2 面向对象设计的原则 …… 98
6.5.3 面向对象设计的任务 …… 98
6.5.4 面向对象设计的处理阶段 …… 98
6.5.5 面向对象设计的过程和步骤 …… 99
6.6 OOP 技术的应用 …… 100
6.7 疑难问题解析 …… 103
职场点拨——团队精神的重要性 …… 103

第二篇 核心技术篇

第 7 章 方法 …… 105
7.1 定义和使用方法 …… 105
7.1.1 方法的返回值 …… 107
7.1.2 方法参数简介 …… 107
7.2 变量作用域 …… 109
7.3 静态方法与实例方法 …… 110
7.4 方法参数详解 …… 113

7.4.1 使用值参数 ……113
7.4.2 使用引用参数 ……114
7.4.3 使用输出参数 ……115
7.4.4 使用参数数组 ……116
7.4.5 使用数组参数 ……117
7.5 结构方法 ……120
7.6 方法重载 ……120
7.6.1 方法重载介绍 ……121
7.6.2 方法重载的具体应用 ……123
7.7 使用 Main()方法 ……126
7.8 递归 ……130
7.9 疑难问题解析 ……131
职场点拨——模块化设计的原则 ……132
第 8 章 类 ……133
8.1 定义类 ……134
8.2 类的成员 ……135
8.2.1 最简单的数据成员 ……135
8.2.2 最重要的函数成员 ……136
8.3 访问修饰符 ……137
8.3.1 类成员访问修饰符 ……137
8.3.2 类的访问修饰符 ……138
8.4 对象是什么 ……139
8.4.1 创建对象 ……139
8.4.2 使用对象 ……140
8.5 C#类成员 ……143
8.5.1 类成员概述 ……143
8.5.2 类成员使用实例 ……143
8.6 使用构造函数 ……145
8.6.1 构造函数概述 ……145
8.6.2 构造函数的参数 ……146
8.6.3 使用私有构造函数 ……146
8.6.4 使用静态构造函数 ……147
8.7 使用析构函数 ……149
8.8 使用只读字段 ……151
8.9 使用 this 关键字 ……151
8.9.1 进行 this 访问 ……151
8.9.2 使用 this 关键字 ……152
8.10 C#属性 ……154
8.11 索引器 ……157

8.12 命名空间······159
8.12.1 编译单元······159
8.12.2 完全限定名标识······160
8.12.3 using 指令······161
8.13 C#内部类······162
8.14 C#分部类······163
8.15 疑难问题解析······164
职场点拨——谈职业精神······164
第 9 章 集合和数组应用······167
9.1 C#集合······167
9.1.1 C#集合概述······168
9.1.2 使用集合······169
9.2 数组的高级应用······170
9.2.1 数组的方法和属性······170
9.2.2 CreateInstance()方法······175
9.2.3 数组转换······178
9.3 C#集合类······180
9.3.1 ArrayList 集合类······180
9.3.2 Hashtable 集合类······183
9.3.3 Queue 集合类······186
9.3.4 SortedList 集合类······188
9.3.5 Stack 集合类······191
9.4 使用 foreach 循环语句······193
9.4.1 foreach 概述······193
9.4.2 集合内使用 foreach 语句······194
9.5 自定义集合类······195
9.5.1 实现接口······195
9.5.2 结合实现······197
9.6 C#迭代器······199
9.7 疑难问题解析······201
职场点拨——老板喜欢什么样的员工······202
第 10 章 继承与多态······203
10.1 继承······204
10.1.1 类的层次结构······204
10.1.2 声明继承······205
10.1.3 继承规则······205
10.2 C#的隐藏······207
10.3 C#的多态······208
10.3.1 虚方法和虚方法重写······209

10.3.2 重写方法的特点 ······210
10.3.3 多态的应用 ······211
10.4 使用 base 关键字 ······214
10.4.1 base 访问 ······214
10.4.2 base 构造函数指定 ······215
10.5 抽象类和抽象方法 ······217
10.6 密封类和密封成员 ······219
10.7 C#静态类 ······221
10.8 C#接口 ······223
10.8.1 定义接口 ······223
10.8.2 接口的实现和继承 ······225
10.8.3 显式接口的成员 ······226
10.8.4 访问接口成员 ······229
10.9 C#根类 object ······231
10.9.1 object 的方法 ······232
10.9.2 object 的方法应用 ······232
10.9.3 Finalize()特殊应用 ······234
10.10 C#类转换 ······235
10.10.1 隐式转换 ······235
10.10.2 显式转换 ······236
10.10.3 使用 is 运算符 ······237
10.10.4 使用 as 运算符 ······239
10.11 C# 4.0 新特性——dynamic 关键字 ······240
10.12 疑难问题解析 ······242
职场点拨——兼职 ······243
第 11 章 字符串与正则表达式 ······245
11.1 C#字符串 ······245
11.1.1 C#类型 ······246
11.1.2 string 和 String ······246
11.1.3 StringBuilder ······253
11.2 C#正则表达式 ······255
11.2.1 语言 ······256
11.2.2 正则表达式类 ······258
11.3 正则表达式应用 ······262
11.3.1 分隔符分隔 ······262
11.3.2 字符检索处理 ······263
11.3.3 字符替换处理 ······264
11.4 C# 4.0 新特性——命名可选参数与 COM 互操作 ······264
11.4.1 命名与可选参数 ······264

11.4.2 可选参数与重载决策 ······265
11.4.3 COM 互操作 ······266
11.5 疑难问题解析 ······267
职场点拨——部门间的沟通 ······268

第三篇 提 高 篇

第 12 章 委托与事件 ······269
12.1 C#委托 ······270
12.1.1 声明委托 ······270
12.1.2 使用委托 ······272
12.1.3 使用委托匿名方法 ······275
12.2 C#事件 ······277
12.2.1 声明事件 ······277
12.2.2 使用事件 ······278
12.2.3 采用访问器方式使用事件 ······279
12.3 C＃4.0 新特性——动态编程 ······281
12.3.1 再议 dynamic ······281
12.3.2 动态编程的应用 ······283
12.4 疑难问题解析 ······286
职场点拨——跳槽的成本 ······286
第 13 章 泛型 ······287
13.1 泛型概述 ······287
13.1.1 泛型的特点 ······288
13.1.2 泛型的优点 ······288
13.2 泛型声明 ······289
13.2.1 声明泛型类 ······289
13.2.2 泛型接口 ······291
13.2.3 泛型委托 ······292
13.2.4 泛型方法 ······293
13.3 泛型约束 ······294
13.4 泛型集合类 ······297
13.4.1 泛型集合类介绍 ······297
13.4.2 泛型集合类的使用 ······297
13.4.3 自定义泛型集合类 ······303
13.5 C#泛型迭代器 ······306
13.6 疑难问题解析 ······308
职场点拨——认清你的职责 ······309
第 14 章 文件操作 ······310
14.1 文件管理 ······311

14.1.1 使用 File 和 FileInfo ······311
14.1.2 使用 Directory 和 DirectoryInfo ······315
14.1.3 使用 Path ······319
14.1.4 使用 Environment ······322
14.2 C#流文件管理 ······325
14.2.1 使用 FileStream ······326
14.2.2 使用 StreamReader 和 StreamWriter ······330
14.2.3 使用 BinaryReader 和 BinaryWriter ······333
14.3 XML 文件处理 ······336
14.3.1 XML 简介 ······336
14.3.2 基于流的处理 ······337
14.3.3 基于 DOM 的处理 ······348
14.4 疑难问题解析 ······350
职场点拨——程序员的几个坏习惯 ······351
第 15 章 Windows 窗体编程 ······353
15.1 窗体编程基础 ······353
15.1.1 窗体概述 ······354
15.1.2 一个简单的窗体程序 ······355
15.2 创建和使用窗体 ······359
15.2.1 创建窗体 ······359
15.2.2 窗体设置 ······360
15.3 使用控件和组件 ······369
15.3.1 控件的属性 ······369
15.3.2 使用 Label 控件和 LinkLabel 控件 ······370
15.3.3 使用 TextBox 控件 ······373
15.3.4 使用 Button 控件 ······376
15.3.5 使用 CheckBox、RadioButton 和 GroupBox 控件 ······379
15.3.6 使用 ListBox 和 ComboBox 控件 ······384
15.3.7 使用 ProgressBar 和 BackgroudWorker ······384
15.3.8 使用 ListView 控件 ······390
15.3.9 使用 TreeView 控件 ······391
15.4 使用窗体菜单 ······392
15.4.1 主菜单实现控件介绍 ······392
15.4.2 使用上下文菜单控件 ······393
15.5 工具栏应用 ······394
15.5.1 ToolStrip 控件介绍 ······394
15.5.2 使用 ToolStrip 控件 ······395
15.6 窗体对话框 ······399
15.6.1 自定义实现对话框 ······399

15.6.2 使用对话框控件…405
15.7 疑难问题解析…410
职场点拨——初次见面要留下好印象…414
第 16 章 Web 应用编程…415
16.1 Web 编程概述…415
16.2 创建一个简单的 ASP.NET 程序…416
16.3 ASP.NET 控件…419
16.3.1 Web 服务器控件…420
16.3.2 数据处理控件…423
16.3.3 其他控件…424
16.4 状态管理…427
16.4.1 ViewState…428
16.4.2 Cookies…428
16.4.3 服务器端状态管理…429
16.5 安全性…431
16.6 ASP.NET 和数据库的连接…433
16.7 疑难问题解析…437
职场点拨——有错要勇于面对同事…437
第 17 章 数据库和 ADO.NET 操作…438
17.1 什么是 ADO.NET…438
17.1.1 数据库简介…439
17.1.2 ADO.NET 的目的…440
17.2 ADO.NET 的对象…441
17.2.1 提供者对象介绍…441
17.2.2 用户对象介绍…442
17.2.3 System.Data 命名空间…442
17.3 安装 Northwind…443
17.4 使用 DataReader…444
17.5 数据库连接…446
17.5.1 基本连接处理…447
17.5.2 连接的使用…447
17.5.3 事务处理…449
17.6 命令和存储过程…450
17.6.1 处理命令…450
17.6.2 存储过程…454
17.7 DataSet 数据处理…457
17.7.1 数据表…457
17.7.2 数据列…457
17.7.3 数据行…458

17.7.4 DataSet 数据更新……460
17.8 ADO.NET 和 XML……463
17.8.1 WriteXml()方法……463
17.8.2 ReadXml()方法……465
17.9 ADO.NET 和 SQL……467
17.9.1 更新操作命令……467
17.9.2 直接执行 SQL 命令……468
17.10 疑难问题解析……470
职场点拨——寻找更好的工作……472
第 18 章 DataGrid 和数据绑定……474
18.1 通过 Visual Studio 2010 实现数据库项目……474
18.1.1 建立数据库连接……475
18.1.2 建立数据源……476
18.2 DataGrid 控件……478
18.2.1 DataGrid 基础……478
18.2.2 使用 DataGridView……480
18.3 数据绑定……488
18.3.1 单一绑定……488
18.3.2 数据绑定对象……488
18.4 疑难问题解析……497
职场点拨——程序员创业的致命错误……497
第 19 章 GDI+图形图像编程……499
19.1 GDI+基础……499
19.1.1 GDI+……500
19.1.2 Graphics 类……500
19.2 图形操作……501
19.2.1 绘图工具……501
19.2.2 绘图……502
19.2.3 颜色混合……505
19.3 测量坐标和区域……508
19.3.1 Point 和 PointF……508
19.3.2 Size 和 SizeF……509
19.3.3 Rectangle 和 RectangleF……510
19.4 字体和文本……511
19.4.1 字体处理……511
19.4.2 文本处理……513
19.5 颜色操作……514
19.5.1 红绿蓝值……515
19.5.2 命名的颜色……515

19.6 图像操作……515
19.6.1 位图文件简介……515
19.6.2 矢量图文件简介……517
19.6.3 GDI+图像处理……518
19.7 打印操作……524
19.8 制作一个画板工具……529
19.9 疑难问题解析……534
职场点拨——谈加薪和升职……537

第四篇 综合实战篇

第 20 章 俄罗斯方块游戏……539
20.1 系统概述与预览……539
20.1.1 游戏的运作过程……539
20.1.2 项目文件概述……539
20.1.3 实例项目概览……540
20.2 系统窗体界面实现……540
20.2.1 制作游戏窗体……540
20.2.2 窗体元素设置文件……543
20.3 事件处理程序……548
20.3.1 初始设置……549
20.3.2 编写事件处理代码……550
20.4 游戏控制、处理方法……552
20.4.1 初始设置……552
20.4.2 重新开始处理……553
20.4.3 Review()处理……554
20.4.4 载入处理……555
20.4.5 绘制方块处理……558
20.4.6 游戏过程处理……561
参考文献……571

第一篇　基　础　篇

第1章　C#开发基础

C#语言是当前最热门的编程语言之一，它继承了 C 和 Java 等语言的优点，引领了编程行业开创性的变革。C#功能强大，它不但能够实现窗体编程，而且也能迅速实现 Web 编程。本章将引导读者逐步进入 C#世界，使读者逐渐掌握 C#语言的基本知识。通过本章能够学到如下知识。

- 什么是 C#。
- .NET Framework 框架。
- 职场点拨——C#的地位。

背景介绍

小菜，21 岁，一名即将毕业的大四学生，正准备开始自学 C#。

Wisdom，小菜的表哥，一名资深软件工程师。

2005 年 X 月 X 日，天气阴

今天老师建议我们选修 C#语言。在过去的 3 年里，我已经学过了 C 语言和 C++，现在开始学习 C#，这对我以后就业还有用吗？



一问一答

小菜：我在大二和大三时期已经学过了 C 语言和 C++，现在大四才开始学习 C#，还有用吗？

Wisdom：C#是微软公司发布的一种面向对象的、运行于.NET Framework 之上的高级程序设计语言。从表面看来，C#与 Java 有着相同的特点，如包括类单一继承和界面独立性的特点，并且和 Java 拥有几乎相同的语法。但是 C#与 Java 也有明显的不同，它与 COM（组件对象模型）是直接集成的，而且它是微软公司.NET Windows 网络框架的主角。现在微软公司正在大力推广它，希望能够使其成为软件开发市场的主流力量。

小菜：C#和 Java、C++相比，究竟哪个更加强大？

Wisdom：在这里我也不能给你具体的答案，只能说各有各的优势。在本章的最后，我会给你简单讲解 C#的地位。

1.1 什么是 C#

C#读作 C Sharp，是从 C 和 C++进化而来的新一代编程语言。在本节的内容中，将简要介绍 C#语言的基本知识。

1.1.1 C#的推出背景

在 C#推出之前，C 和 C++是在商业软件的开发领域中使用最广泛的语言。C 和 C++为程序员提供了十分灵活的操作，但同时也牺牲了一定的效率。例如和 Visual Basic 等语言相比，往往需要更长的时间来开发同等级别的 C/C++应用程序。由于 C/C++语言的复杂性，许多程序员都试图寻找一种新的语言，希望能在功能与效率之间找到一个更为理想的平衡点。

有很多语言是以牺牲灵活性为代价来提高效率的，而 C/C++程序员非常需要这些灵活性。往往解决方案对编程人员的限制过多（如屏蔽一些底层代码控制的机制），其提供的功能会差强人意。这些语言无法方便地同早先的系统交互，也无法很好地和当前的网络编程相结合。

对 C/C++开发者来说，最希望的解决方案是能够在快速开发的同时又可以调用底层平台的所有功能，有一种和最新的网络标准保持同步并且能和已有的应用程序良好整合的环境。另外，一些 C/C++开发人员还需要在必要的时候进行一些底层的编程。

针对上述问题，微软公司推出了 C#解决方案。这是一种最新的、面向对象的编程语言。它使得程序员可以快速地编写各种基于 Microsoft 的.NET 平台的应用程序。Microsoft 的.NET 提供了一系列的工具和服务，方便开发软件项目。

无论是高级的商业对象，还是系统级的应用程序，正是因为 C#面向对象的卓越设计和优秀表现，成为了构建各类组件的最佳选择。最重要的是，C#使得 C++程序员可以高效地开发程序，而绝不损失 C/C++原有的强大功能，并且因为继承关系，C#与 C/C++具有极大的相似性。熟悉类似语言的开发者可以很容易学习 C#。

1.1.2 C#的特点

C#语言的定义主要是从 C 和 C++继承而来的，而且语言中的许多元素也反映了这一点。和 Java 相比，C#从 C++继承的可选选项方面更要广泛一些，并且 C#还增加了自己独有的新特点。C#的特点主要体现在如下四个方面。

1．从 Java 继承

C#从 Java 中继承了它的大多数特点，包括实用语法和范围等。例如最基本的“类”，在 C#中类的声明方式和在 Java 中很相似。Java 的关键字 import 在 C#中被替换成了 using，但是它们起到了同样的作用，并且一个类开始执行的起点是静态方法 Main()。

2．从 C 和 C++继承

C#从 C 和 C++继承的特点主要体现在如下三个方面。

- 编译：程序直接编译成标准的二进制可执行形式，但 C#的源程序并不被编译成二进制可执行形式，而是一种中间语言，类似于 Java 字节码。例如，一个名为 Hello.cs 的程序文件将被编译成名为 Hello.exe 的可执行程序。

- 结构体：一个 C#的结构体与 C++的结构体是相似的，因为它能够包含数据声明和方法，但是 C#结构体与类是不同的，并且不支持继承。
- 预编译：C#中存在预编译指令支持条件编译、警告、错误报告和编译行控制。

3．C#独有的特点

和 C、C++和 Java 相比，C#的独有特点主要体现在如下几个方面。

- 中间代码：微软公司在用户选择何时 MSIL（微软中间语言）应该编译成机器码的时候预留了很大的余地。微软公司声称 MSIL 不是解释性的，而是被编译成了机器码。而这种实现方式决定了基于 MSIL 的程序将在性能上超过“解释性的”Java 代码，但是 Java JIT 编译器的普遍存在使得 Java 和 C#在性能上基本相同。所谓的“C#是编译语言而 Java 是解释性的”之说也只是商业技巧。Java 的中间代码和 MSIL 都是中间的汇编形式的语言，它们在运行时或其他的时候被编译成机器代码。
- 命名空间中的声明：当创建一个程序的时候，在一个命名空间里创建了一个或多个类，在这个命名空间里还有可能声明界面、枚举类型和结构体，并必须使用 using 关键字来引用其他命名空间的内容。
- 拥有比 C、C++或 Java 更多的数据类型：例如有 bool、byte、ubyte、short、ushort、int、uint、long、ulong、float、double 和 decimal。这些数据类型都有一个固定的大小，并且每个数据类型都分为有符号类型和无符号类型两种。
- 有两个基类：在 C#中，object 类是所有其他类的基类。另外，string 类也和 object 一样是 C#语言的一部分。无论何时在程序中写入一句带引号的字符串，编译器会创建一个 string 对象来保存它。
- 传递参数：C#能够传递数目可变的参数，默认的参数传递方法是对基本数据类型进行值传递。使用 ref 关键字可以用来强迫一个变量通过引用传递，这使得一个变量可以接受一个返回值。使用 out 关键字也可以声明引用传递过程，它和 ref 不同的是指明这个参数并不需要初始值。
- 与 COM 的集成：COM 是微软公司的 Win32 组件技术，最有可能在任何.NET 语言里编写 COM 客户和服务器端。C#能够实现对 Windows 程序与 COM 的无缝集成。使用 C#编写的类，可以子类化一个已存在的 COM 组件，生成的类也能被作为一个 COM 组件使用，然后又能重复使用。
- 索引下标：一个索引与属性用一个方括号中的数字来匿名引用，而不使用属性名来引用类成员。

4．基本特点

C#的基本特点主要体现在如下几点：

- 简单：和 C++相比，C#的学习更加简单。
- 面向对象：C#支持所有关键的面向对象的概念，例如封装、继承和多态性。完整的 C#类模式构建在 NGWS 运行时的虚拟对象系统的上层，对象模式只是基础的一部分，不再是编程语言的一部分。
- 类型安全：在使用过程中必须遵守 C#中一些相关变量的规则，因为 C#使用了最严格的类型安全来保护自己及垃圾收集器。所以不能使用没有初始化的变量，取消不安全的类型转换，实施边界检查。

- 兼容：C#允许使用 NGWS 的通用语言规定来访问不同的 API。在 CLS 中规定了一个标准，能够实现在符合这种标准的语言的内部之间进行操作。为了加强 CLS 的编译，C#编译器检测所有的公共出口编译，并在不通过时列出错误。
- 灵活：C#在访问原始 Win32 代码时，可能会导致对非安全类指定指针的使用。虽然 C#代码的默认状态是类型安全的，但是可以声明一些类或仅声明类的方法是非安全类型的。这样的声明允许使用指针、结构，并静态地分配数组。

1.2 .NET Framework 框架

微软公司在 2000 年推出了.NET Framework 框架，并将其作为重新树立自己在软件业界的信心和地位的崭新战略和概念。通过.NET 平台，可以使用多种语言开发同一个项目，实现了这些语言的跨平台应用。微软公司为.NET 专门配备了一门新的语言，这便是 C#语言。

1.2.1 .NET Framework 简介

.NET Framework 为开发人员提供了公共语言运行库的运行环境，它能够运行代码并为开发过程提供更轻松的服务。公共语言运行库的功能是，通过编译器和工具公开，开发人员可以编写利用此托管执行环境的代码。托管代码是指使用基于公共语言运行库的语言编译器开发的代码。托管代码具有许多优点，例如跨语言集成、跨语言异常处理、增强的安全性、版本控制和部署支持、简化的组件交互模型、调试和分析服务等。

如果准备使公共语言运行库向托管代码提供服务，语言编译器必须生成一些元数据，这些元数据可以描述代码中的类型、成员和引用。通常将元数据与代码一起存储，每个可加载的公共语言运行库可移植执行文件都包含元数据。公共语言运行库可以使用元数据来完成各种任务，例如常见的查找类和加载类等。

公共语言运行库可以自动处理对象布局，并且能够管理对象引用，当不再使用对象时可以释放它们。按上述方式实现生存期管理的对象被称为托管数据。使用垃圾回收机制，可以消除内存泄露以及其他一些常见的编程错误。如果编写的代码是托管代码，就可以在.NET Framework 应用程序中使用托管数据、非托管数据或同时使用这两种数据。因为语言编译器会提供自己的类型，例如基元类型，所以很可能开发人员并不总是知道（或需要知道）这些数据是否是托管的。

使用公共语言运行库，可以很容易地设计出能够跨语言交互的组件和应用程序。用不同语言编写的对象之间可以互相通信，并且可以紧密集成它们的行为。例如，可以定义一个类，然后使用不同的语言从原始类派生出另一个类或调用原始类的方法。也可以将一个类的实例，传递到用不同的语言编写的另一个类的方法。因为基于公共语言运行库的语言编译器和工具，都使用由公共语言运行库定义的通用类型系统，所以使跨语言集成为了可能，并且它们遵循公共语言运行库关于定义新类型以及创建、使用、保持和绑定到类型的规则。

在.NET 中，所有托管组件都附带生成它们所基于的组件和资源的信息，这些信息成为了元数据的一部分。当在公共语言运行库中使用这些信息后，能够保证组件或应用程序具有它需要的所有内容的指定版本，这样会使代码不太可能因为某些未满足的依赖项而发生中断。另外，因为在注册表中很难建立和维护这些信息，所以注册信息和状态数据不再保存在

注册表中。取而代之的是定义的类型及其依赖项的信息作为元数据与代码存储在一起，这样就大大降低了组件复制和移除任务的复杂性。

当使用语言编译器和工具公开公共语言运行库的功能后，对开发人员来说将更加有用并且直观。也就是说，公共语言运行库的某些功能可能在一个环境中比在另一个环境中更突出。公共语言运行库具有如下主要优点。

- 改进了性能。
- 能够轻松使用用其他语言开发的组件。
- 通过类库提供了可扩展类型。
- 具备高级面向对象的功能，例如面向对象编程的继承、接口和重载；允许创建多线程的可缩放应用程序的显式自由线程处理支持；结构化异常处理和自定义属性支持。

例如，如果使用 Microsoft Visual C++.NET，则可以使用 C++托管扩展来编写托管代码。C++托管扩展提供了托管执行环境，以及对所熟悉的强大功能和富于表现力的数据类型的访问等特点。

其他公共语言运行库功能还包括如下特点。

- 跨语言集成，特别是跨语言继承。
- 垃圾回收，它管理对象生存期，使引用计数变得不再必要。
- 自我描述的对象，它使得使用接口定义语言（IDL）不再是必要的。
- 编译一次即可在任何支持公共语言运行库的 CPU 和操作系统上运行。

另外，还可以使用 C#语言编写托管代码。C#语言提供了下列优点。

- 完全面向对象的设计。
- 非常强的类型安全。
- 很好地融合了 Visual Basic 的简明性和 C++的强大功能。
- 垃圾回收。

虽然 C# 完全符合公共语言规范，但是 C# 本身不具有单独的运行时库。事实上，.NET 框架就是 C# 的运行时库，C# 的编程库是.NET 类库，所以能够使用.NET 框架类库的所有类。C#能够实现下列.NET 框架所支持的全部功能。

- Windows 窗体编程。
- ADO.NET 数据库编程。
- XML 编程。
- ASP.NET 的 Web 编程。
- Web 服务编程。
- 与 COM 和 COM+互操作性编程。
- 通过 P/Invoke 调用 Windows API 和任何动态链接库中的函数。

1.2.2　不得不知的几个概念

在下面的内容中，将简要介绍几个和 C#相关的重要概念。

1．内存分配

在新进程初始化时，运行时会为进程保留一个连续的地址空间区域，这个保留的地址空

间被称为托管堆。托管堆维护着一个指针，用这个指针指向将在堆中分配的下一个对象的地址。开始该指针设置为指向托管堆的基址，在托管堆上部署了所有引用类型。当应用程序创建第一个引用类型时，会为托管堆的基址中的类型分配内存。当应用程序创建下一个对象时，垃圾回收器在紧接第一个对象后面的地址空间内为它分配内存。只要地址空间可用，垃圾回收器就会继续以这种方式为新对象分配空间。

从托管堆中分配内存要比非托管内存分配速度快。由于运行时通过为指针添加值来为对象分配内存，所以这几乎和从堆栈中分配内存一样快。另外，由于连续分配的新对象在托管堆中是连续存储的，所以应用程序可以快速访问这些对象。

2．释放内存

垃圾回收器的优化引擎很重要，它能够根据所执行的分配决定执行回收的最佳时间。垃圾回收器在执行回收时，会释放应用程序不再使用的对象的内存。它通过检查应用程序的根来确定不再使用的对象。每个应用程序都有一组根。每个根或者引用托管堆中的对象，或者设置为空。应用程序的根包含全局对象指针、静态对象指针、线程堆栈中的局部变量和引用对象参数以及 CPU 寄存器。垃圾回收器可以访问由实时（JIT）编译器和运行时维护的活动根的列表。垃圾回收器对照此列表检查应用程序的根，并在此过程中创建一个图表，其中包含所有可从这些根中访问的对象。

3．.NET Framework 类库

.NET Framework 包括三部分，分别是类、接口和值类型。它们可以加速和优化开发过程并提供对系统功能的访问。为了便于语言之间进行交互操作，.NET Framework 类型是符合 CLS 的，所以可以在任何编程语言中使用，只要这种语言的编译器符合公共语言规范（CLS）。

.NET Framework 类型是生成 .NET 应用程序、组件和控件的基础。.NET Framework 中的类能够实现如下功能。

1）表示基础数据类型和异常。

2）封装程序中的数据结构。

3）执行 I/O 操作。

4）访问和加载类型相关的信息。

5）调用.NET Framework 安全检查。

6）提供数据访问、多客户端 GUI 和服务器控制的客户端 GUI。

.NET Framework 提供一组丰富的接口以及抽象类和具体（非抽象）类。可以按原样使用这些具体的类，或者在多数情况下从这些类派生自己的类。若要使用接口的功能，既可以创建实现接口的类，也可以从某个实现接口的 .NET Framework 类中派生类。

1.3 疑难问题解析

本章详细介绍了 C#开发的基本知识。本节中，将对本章中比较难以理解的问题进行讲解。

读者疑问：什么是命名约定？

解答：.NET Framework 类型使用点语法命名方案，该方案隐含了层次结构。此技术将

相关类型分为不同的命名空间组，以便可以更容易地搜索和引用它们。全名的第一部分（最右边的点之前的内容）是命名空间名，全名的最后一部分是类型名。例如，System.Collections.ArrayList 表示 ArrayList 类型，该类型属于 System.Collections 命名空间。System.Collections 中的类型可用于操作对象集合。

此命名方案使扩展.NET Framework 的库开发人员可以轻松创建分层类型组，并用一致的、带有提示性的方式对其进行命名。利用命名模式将相关类型分组为命名空间是生成和记录类库的一种非常有用的方式。但是，此命名方案对可见性、成员访问、继承、安全性或绑定无效。一个命名空间可以被划分在多个程序集中，而单个程序集可以包含来自多个命名空间的类型。程序集为公共语言运行库中的版本控制、部署、安全性、加载和可见性提供外形结构。

读者疑问：什么是系统命名空间？

解答：System 命名空间是.NET Framework 中基本类型的根命名空间。此命名空间包括表示由所有应用程序使用的基础数据类型的类：Object（继承层次结构的根）、Byte、Char、Array、Int32、String 等。在这些类型中，有许多与编程语言所使用的基元数据类型相对应。当使用 .NET Framework 类型编写代码时，可以在应使用 .NET Framework 基础数据类型时使用编程语言的相应关键字。



职场点拨——C#的地位

在武侠小说中，令人们热血澎湃的剑客都是仗剑走天涯。剑不离手，手不离剑。作为一名程序员，在入行之前也都会根据自己的目标去选择自己的武器——编程语言。当今编程语言种类繁多，作为立志要步入编程行业的读者们，你们选择哪一把绝世好剑呢？

Visual C++、Visual Basic、Delphi、Java，这些耀眼的名字占据了程序员的生活。微软公司为了继续在开发市场占据重要位置，在 2000 年开始推出了.NET 战略。作为.NET 的核心开发语言，C#顺理成章地浮出了水面。如果抛开一切非技术方面的因素，C#无疑是最全面的编程语言，它几乎集中了所有关于软件开发和软件工程研究的最新成果：面向对象、类型安全、组件技术、自动内存管理、跨平台异常处理、版本控制、代码安全管理。表 1-1 为国外某编程社区进行的 2009 年和 2010 年编程语言使用率统计表。

表 1-1 2009 年和 2010 年编程语言使用率统计

2010 年排名	2009 年排名	编 程 语 言	2010 年占有率(%)	和 2009 年相比(%)
1	2	C	18.058	+2.59
2	1	Java	18.051	–1.29
3	3	C++	9.707	–1.03
4	4	PHP	9.662	–0.23
5	5	Visual Basic	6.392	–2.70
6	6	C#	4.435	+0.38

由表中的统计数据可以看出，虽然C#是一种新兴的编程语言，但是排名却比较靠前。

下面来看C#的主要应用领域。

（1）Web领域

当前基于C#的ASP.NET技术已经和JSP、PHP并列为三大Web开发技术，并且伴随着Visual Studio.NET版本的不断更新换代，其优势越来越明显。ASP.NET完全支持C#语言和Visual Basic，两者之间可以相互切换，并且利用Visual Studio.NET，可以很好地和其他的微软产品结合使用，例如SQL Server。

（2）桌面领域

Visual Studio.NET功能强大，不但能开发Web程序，而且还能开发窗体程序，并且其开发界面和Visual Basic、MFC一样简单易用。微软公司已经推出了Visual Studio.NET，它和Visual Basic完全兼容，Visual Studio.NET+C#在桌面的应用已经能够和Visual C++ 6.0分庭抗礼。

（3）WPF

C#可以开发出WPF程序。利用Visual Studio.NET工具，C#能够实现和Vista界面一样的绚丽效果，吸引广大用户的眼球。

由此可见，C#作为一种新的编程语言，它融合了几种语言的优点，整合了它们的功能，并推出了先进的配套开发工具Visual Studio.NET，因此肯定会拥有一个美好的前景。

第 2 章　初识 C#开发环境

C#语言是一种功能强大的综合性开发语言，是建立在特定开发平台上的。在进行 C#开发前，需要为其建立专门对应的开发平台，搭建开发环境。本章的将简要介绍 C#开发环境的搭建方式，并简要讲解 Visual Studio 2010 可视化集成开发环境的基本知识。通过本章能学到如下知识。

- ❑ Visual Studio.NET 2010 介绍。
- ❑ 初步认识 Visual Studio 2010 开发环境。
- ❑ 职场点拨——怎样学好 C#。

2005 年 X 月 XX 日，天气阴

学习 C#已经有一段时间了，我感觉 C#比较难学，要想完全掌握还需要耗费很多的时间和精力。我现在对学好它没有信心，希望有一个快速学习 C#的好方法。



一问一答

小菜：感觉 C#好难啊，能不能告诉我一种快速学习 C#的方法或者捷径？

Wisdom：学习编程没有捷径，急功近利反而会欲速则不达。你应该端正学习态度，在本章最后我将重点介绍学习 C#的方法。本章将要学习搭建 C#开发环境。古人云：工欲善其事，必先利其器。作为一个 C#初学者的你，选择合适的开发工具比较重要。有了好的工具就能提高学习效率，这可以勉强作为初学者学习 C#的一条捷径吧。

小菜：我应该用什么工具？

Wisdom：微软为 C#推出的功能强大的配套开发工具——Visual Studio，当前的最新版本是 Visual Studio 2010。

2.1　Visual Studio 2010 介绍

本节简要介绍 Visual Studio 2010 开发工具的基本知识。为读者步入后面知识的学习，打下坚实的基础。

2.1.1　Visual Studio 2010 工具介绍

Visual Studio 2010 是微软公司推出的全新专用开发工具，它是一个集成的开发环境工

具，能够应用于 Visual Basic 2010、Visual C++ 2010、Visual C# 2010 和 Visual J# 2010 等专业编程语言。通过 Visual Studio 2010 可以在同一项目内使用不同的语言，并能实现它们相互间的无缝接口处理，共同实现指定的功能。

微软产品 Visual Studio 的推出被称为软件行业的革命，而 Visual Studio 2010 是当前最新的版本，为这个跨时代的产品提供了强有力的实现支持，为程序员的学习、使用带来了极大的方便。所以 Visual Studio 一经推出便受到了用户的欢迎，并且随着技术的成熟，各种版本也相继被推出。

2.1.2 Visual Studio 2010 工具的内容

因为 Visual Studio 是一个集成的开发环境工具，所以包括了大量的工具平台供用户调用和设置。Visual Studio 包括了多个集成开发所必需的工具，具体内容如下。

1．Visual J#

Visual J#是一种开发工具，供熟悉 Java 语法的开发人员在.NET Framework 上生成应用程序和服务时使用。该工具将 Java 语言的语法集成到 Visual Studio 集成开发环境（IDE）中。Visual J#还支持 Visual J++ 6.0 中所具有的大多数功能，包括 Microsoft 扩展。Visual J#不是一种用于开发在 Java 虚拟机上运行的应用程序的工具。使用 Visual J#生成的应用程序和服务只在 2010 Framework 上运行。Visual J# 由微软公司独立开发。

2．智能设备应用程序

Visual Studio 集成开发环境现在包括开发智能设备（如 Pocket PC）应用程序的工具。通过使用这些工具和.NET Framework 4.0 精简版，可以在个人数字助理（PDA）、移动电话和其他资源受约束的设备中，创建、生成、调试和部署在.NET Framework 4.0 精简版上运行的应用程序。

3．Web 窗体

Web 窗体是用于创建可编程 Web 页的 ASP.NET 技术。Web 窗体将自己呈现为浏览器兼容的 HTML 和脚本，这使得任何平台上的任何浏览器都可以查看 Web 页。使用 Web 窗体，通过将控件拖放到设计器上然后添加代码来创建 Web 页，与创建 Visual Basic 窗体的方法相似。

4．Windows 窗体

Windows 窗体是用于 Microsoft Windows 应用程序开发的、基于 2010 Framework 的新平台。此框架提供一个有条理的、面向对象的、可扩展的类集，使用户能够开发功能丰富的 Windows 应用程序。另外，Windows 窗体也可以作为多层分布式解决方案中的本地用户界面。

5．XML Web Services

XML Web Services 可以通过 HTTP 使用 XML 来接收请求和数据的应用程序。因为 XML Web Services 不受特定组件技术或对象调用约定的制约，所以可被任何语言、组件模型或操作系统访问。在 Visual Studio 中，可以使用 Visual Basic、Visual C#、JavaScript、C++的托管扩展或 ATL Server 快速创建和包含 XML Web Services。

6．XML 支持

XML 是可扩展置标语言的缩写，提供了一种描述结构数据的方法。XML 是 SGML 的

子集，非常适合在 Web 上传送。万维网联合会定义了 XML 标准，以确保结构化数据保持统一并独立于应用程序。Visual Studio 完全支持 XML，并且提供了 XML 设计器，能够更容易地编辑 XML 和创建 XML 架构。

7．.NET Framework 4.0

.NET Framework 4.0 是用于生成、部署和运行 XML Web Services 与应用程序的多语言环境，它由如下三个主要部分组成。

（1）公共语言运行库

在组件的运行和开发操作中，运行库都起到了很大的作用。在组件运行时，运行库除了负责满足此组件在其他组件上可能具有的依赖项以外，还能够管理内存分配、启动和停止线程和进程，以及强制执行安全策略。在开发时，运行库的作用稍有变化。因为能够实现内存管理等自动处理工作，所以运行库使开发人员的操作变得非常简单，减少了开发人员为将业务逻辑转变为可重用组件而必须编写的代码量。

（2）统一编程类

.NET Framework 4.0 为开发人员提供了统一的、面向对象的、分层的和可扩展的类库集（API）。当前，C++开发人员使用 Microsoft 基础类，Java 开发人员使用 Windows 基础类。.NET Framework 4.0 框架统一了这些不同的模型，并且为 Visual Basic 和 JavaScript 程序员也提供了对类库的访问。通过创建几乎跨所有编程语言的公共 API 集，公共语言运行库使得跨语言继承、错误处理和调试成为可能。无论是 JavaScript 还是 C++，这些编程语言对框架都有相似的访问，开发人员可以自由选择其要使用的语言。

（3）ASP.NET 4.0

ASP.NET 4.0 建立在.NET Framework 4.0 的编程类之上，提供了一个 Web 应用程序模型，并且包含使生成 Web 应用程序变得简单的控件集和结构。ASP.NET 4.0 包含封装公共 HTML 用户界面元素的控件集，例如文本框和下拉菜单。在服务器端，控件将公开一个面向对象的编程模型，为 Web 开发人员提供了面向对象的编程的丰富性。ASP.NET 4.0 还提供了结构服务机制，例如会话状态管理和进程回收，这样减少了开发人员编写的代码量，并提高了应用程序的可靠性。另外，使用 XML Web Services 功能，ASP.NET 4.0 开发人员可以编写自己的业务逻辑，并使用 ASP.NET 4.0 结构通过 SOAP 交付该服务。

2.1.3　Visual Studio 2010 新特性

Visual Studio 2010 是最新的一个版本，它可以自定义开始页，并包括以下新功能。

- C# 4.0 中的动态类型和动态编程。
- 支持多显示器。
- 使用 Visual Studio 2010 的特性支持 TDD。
- 支持 Office。
- 具有 Quick Search 特性。
- 新增了 C++ 0x 特性。
- 增强了 IDE。
- 使用 Visual C++ 2010 创建 Ribbon 界面。
- 新增了基于.NET 平台的 F#语言。

而根据微软公司发布的一份官方文档宣称，Visual Studio 2010 和.NET Framework 4.0 在以下 5 个方面有所创新。

（1）民主化的应用程序生命周期管理

在一个组织中，应用程序生命周期管理（ALM）将牵涉多个角色，但实际上，这一过程中的每个角色并不是完全平等的。Visual Studio 2010 坚持打造一个功能平等、共同分担的平台以用于组织内的应用程序生命周期管理过程。

（2）顺应新的技术潮流

每年，业界内的新技术和新趋势层出不穷。通过 Visual Studio 2010，微软公司为开发者提供了合适的工具和框架，以支持软件开发中最新的架构、开发和部署。

（3）让开发商惊喜

从 Visual Studio 的第一个版本开始，微软公司就将提高开发人员的工作效率和灵活性作为自己的目标。Visual Studio 2010 继续关注并且显著地改进了开发者最核心的开发体验。

（4）下一代平台浪潮的弄潮儿

微软公司将继续投资于市场领先的操作系统，工具软件和服务器平台，为客户创造更高的价值。使用 Visual Studio 2010，可以在新一代的应用平台上，为开发人员的客户创造令人惊奇的解决方案。

（5）跨部门的应用

客户可以在不同规模的组织内创建应用，跨度从单个部门到整个企业。Visual Studio 2010 可以确保在这么宽泛的范围内的应用开发都得到支持。

2.1.4 Visual Studio 2010 与 2008 版本的对比

和 Visual Studio 2008 相比，Visual Studio.NET 2010 有如下突出的优点。

（1）灵活高效的全新 IDE

自从微软公司于 1998 年发布 Visual Studio 6 以来，Visual Studio 的 IDE 已经成为软件开发工具的标杆，很多其他的开发工具，甚至是其他用途的应用程序，都在模仿 Visual Studio 的 IDE。但是，从 Visual Studio 6 到 Visual Studio 2008，虽然 IDE 的功能越来越多，但并没有什么革命性的变化，反倒因为功能太多带来了使用上的不便，导致开发效率低下。程序员们都在期盼一个全新的 IDE 的出现。

现在，几乎程序员们的所有梦想在 Visual Studio 2010 中都成为了现实。在 Visual Studio 2010 中，微软公司用全新的 WPF 技术重新打造了它的编辑器。借助 WPF 的强大功能，新的编辑器可以实现很多以前 Visual Studio 2008 的 IDE 根本无法想象的功能，比如代码的无级缩放，多窗口即时更新，文档地图，代码的自动产生，等等。这些新的 IDE 特性都会极大地提高程序员的开发效率。

（2）云计算

“云计算”的基本原理是，通过使计算分布在大量的分布式计算机上，而不是本地计算机或远程服务器中，企业数据中心的运行将更与互联网相似。这使得企业能够将资源切换到需要的应用上，根据需求访问计算机和存储系统。

早在“云计算”的概念刚刚兴起的时候就有传言，微软将进军“云计算”，将旗下的软件业务纳入“云计算”。现在随着 Visual Studio 2010 CTP 和 Windows Azure 的发布，这一切

成为了事实。Windows Azure 是一个托管服务套件，包括了虚拟计算、可扩展存储以及自动化服务管理系统等。这些工具用来支持微软提供的服务，例如 MSN、Xbox Live 以及 Office Online 等，实现服务的网络化。

对于开发者而言，Windows Azure 已经建立起了一个简单而快速的系统，并且它拥有着标准的模型，这是步入“云计算”时代的捷径。开发者可以根据自己的需要选择第二层功能来使用，比如数据库、业务工具甚至是第三方软件提供的功能。通过 Windows Azure 云计算平台，为开发者提供了灵活性和可开发性，同时还需要考虑利用目前现有的技能、工具和技术，比如微软.NET 框架和 Visual Studio。使用 Windows Azure Tools for Visual Studio，可以为 Windows Azure 创建、调试和部署服务与应用程序。Visual Studio 2010 为 Windows Azure 提供了专门的项目模型，同时也可以利用 Visual Studio 2010 对服务和应用程序进行调试。另外，还可以利用 Visual Studio 2010 将创建的服务打包，然后通过 Windows Live Developer Portal 部署到 Windows Azure。

总之，有了 Visual Studio 2010 的帮助，就可以拔得“云计算”的头筹。

（3）并行计算

Visual Studio IDE 对并行计算开发提供了大力支持。比如，Visual Studio 2010 的调试器知道代码的并行特性，并且能够在调试程序的不同执行单元表现应用程序的状态。

非托管的 C++库和编译器对并行计算的支持，.NET Framework 4.0 对并行计算的大力支持，包括 P-LINQ、并行语言语句等。

另外，Visual Studio 2010 还提供了一个并行性能分析器，它可以用来分析应用程序的性能瓶颈，找到需要并行处理和可以进行并行处理的地方，并以图形化的形式表现出来。这样，并行性能分析器配合 Visual Studio 2010，就可以轻松地实现应用程序的并行化，再次吃上“免费的午餐”。

（4）C++王者归来

自从 Visual Studio 6 推出以后，Visual Studio 中的 C++再也没有多大的变化，这些版本都只是对 C++进行一些细小的改善。但是随着这次 C++新标准 C++0x 的公布，Visual Studio 2010 在 C++开发方面也有了很多革命性的变化。

首先是对 C++新标准 C++0x 的全面支持，不会再像 Visual C++ 6 那样对 C++标准支持不力。在 IDE 方面，微软公司将 Visual C++的构建系统 VCBuild 整合到了 MSBuild 中。借助后台编译后，Visual C++的 IntelliSense 将更加智能，能够处理更多的文件，更加复杂的项目。另外在 MFC 方面，通过引入很多新的类，MFC 开始全面支持 Vista、Windows 7 风格的 UI。这些特性，都成为 Visual Studio 2010 与 Visual Studio 2008 的一个重要差别。相信 C++程序员都会选择 Visual Studio 2010 而略过 Visual Studio 2008。

（5）面向 Windows 7

Visual Studio 2008 是基于 Vista 平台的，现在微软最新的平台是 Windows 7。作为面向下一代平台的开发工具，Visual Studio 2010 提供了很多工具来帮助开发者开发基于 Windows 7 的应用程序，同时使那些已经存在的非托管应用程序，通过一定的处理也同样能够具有新的操作系统所带来的特性。在 Visual Studio 2010 中，微软公司花了很大的力气来使得非托管 C++代码的开发更加容易和高效。例如，升级了 MFC 的库和头文件以全面支持 Windows 7 的界面元素，包括对 Ribbon 界面、搜索功能甚至多点触摸特性的支持。

2.1.5 安装 Visual Studio 2010

Visual Studio 2010 有不同的版本，具体说明如下。

- Visual Studio 2010 Ultimate 企业旗舰版，全功能版本，包含 Visual Studio 2010 Premium 企业版及 Visual Studio 2010 Professional 专业版的所有功能；从需求管理到架构设计、开发、测试、部署、专案管理等涵盖软件开发生命周期管理的所需功能，取代上一版 Visual Studio Team Suite。
- Visual Studio 2010 Premium 企业版，进阶程序开发、查错及测试工具，取代上一版 Visual Studio Team Development 及 Visual Studio Team Database。
- Visual Studio 2010 Professional 专业版，基本程序开发工具。
- Visual Studio Test Professional 2010 专业测试版，测试管理及手动测试执行。
- Visual Studio Lab Management 2010 测试实验室版，运用 Hyper-V 虚拟化技术建立和配置软件测试实验室。
- Visual Studio Team Foundation Server 2010，专案流程管理、版本管控及协同开发的基础平台。此新版本加强了安装模式的选择，取代并转换旧一代 Visual SourceSafe 管控工具。
- Visual Studio Team Explorer Everywhere 2010，异质平台开发管理，让 UNIX/Linux/Mac 移植平台开发及 Java 开发人员，也可纳入 Team Foundation Server 的管理。

硬件要求：最好具有酷睿 II 2.0GHz 以上的 CPU，至少应有 2GB 的 RAM 内存，其中 1GB 用于维持操作系统。

Visual Studio 2010 的具体安装步骤如下。

1）将安装盘放入光驱，或双击存储在硬盘内的安装文件 autorun.exe，弹出安装界面，如图 2-1 所示。

2）单击“安装 Microsoft Visual Studio 2010”选项，弹出组件加载界面，如图 2-2 所示。



图 2-1 开始安装界面



图 2-2 组件加载界面

3）加载完毕单击“下一步”按钮，弹出“起始页”界面，选中“我已阅读并接受许可条款”单选按钮，如图 2-3 所示。

4）单击“下一步”按钮，弹出“选项页”界面，选中“完全”单选按钮，选择安装路

径，如图 2-4 所示。

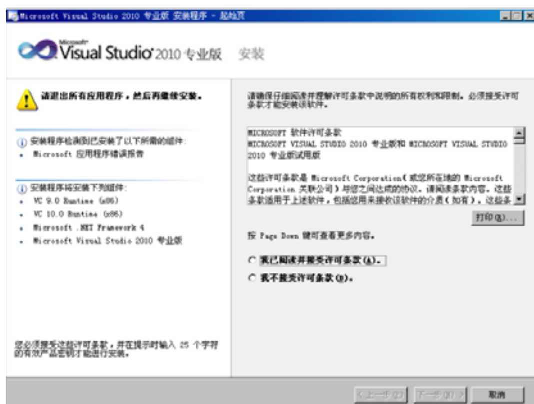


图 2-3　“起始页”界面



图 2-4　“选项页”界面

5）单击“安装”按钮后弹出“安装页”界面，开始进行安装，如图 2-5 所示。

6）安装完毕弹出“完成页”界面，Visual Studio 2010 成功完成安装，如图 2-6 所示。

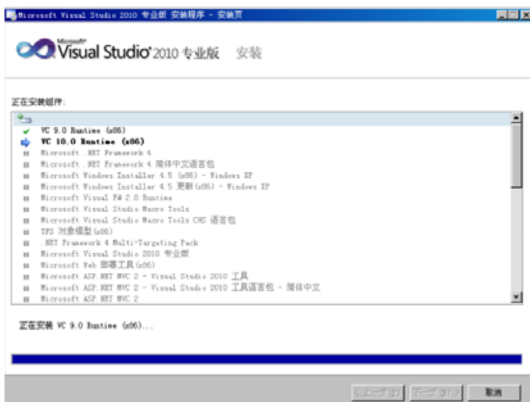


图 2-5　“安装页”界面



图 2-6　“完成页”界面



注意

在图 2-5 的安装过程中，会多次重新启动，重启后将自动进入安装界面。在进行安装时，读者不要惊慌。

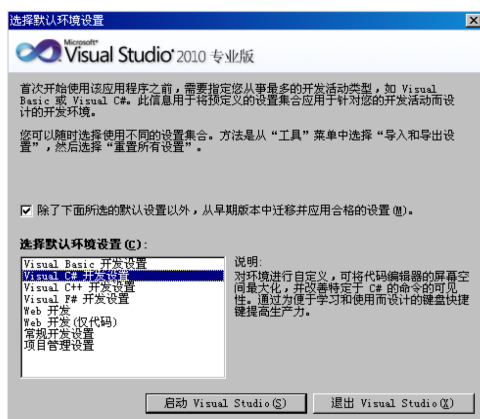
2.2　初步认识 Visual Studio 2010 开发环境

安装 Visual Studio 2010 完毕，需要对其进行设置，使其符合自己的开发需求。本节将简要介绍 Visual Studio 2010 的设置方法，并对其开发环境进行简要介绍和说明。

2.2.1　Visual Studio 2010 设置

首次打开安装后的 Visual Studio 2010，将弹出“选择默认环境设置”对话框。因为本书讲解的是 C#开发，所以选择“Visual C#开发设置”选项，如图 2-7 所示。

然后单击“启动 Visual Studio”按钮后开始配置环境，如图 2-8 所示。



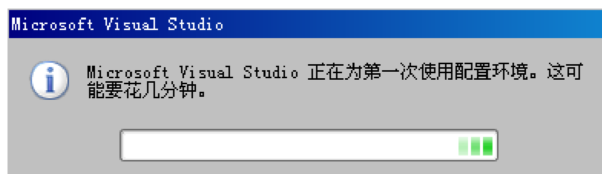


图 2-7 “选择默认环境设置”对话框　　　　图 2-8 配置环境

配置环境完成后将弹出 Visual Studio 2010 的默认集成开发界面，如图 2-9 所示。



图 2-9 Visual Studio 2010 的默认集成开发界面

2.2.2 新建项目

通过 Visual Studio 2010 可以迅速地创建一个项目，包括 Windows 应用程序、控制台程序和 Web 应用程序等常用项目。其方法是在菜单栏中依次单击“文件”|“新建”|“项目”命令，弹出“新建项目”对话框，在此可以设置项目的类型，如图 2-10 所示。

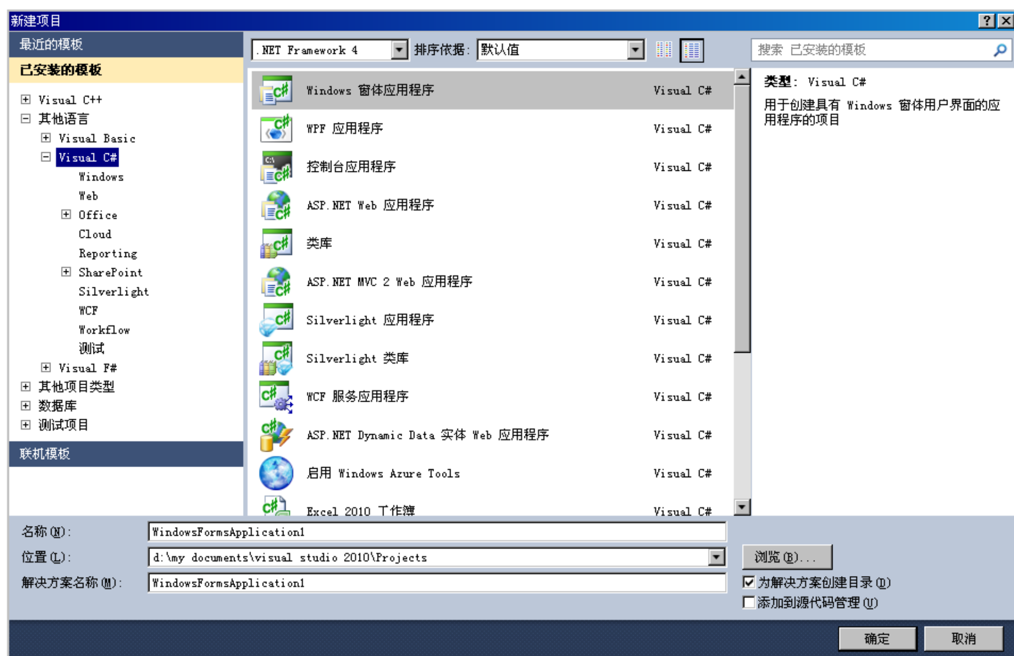


图 2-10　“新建项目”对话框

在菜单栏中依次单击“文件”|“新建”|“网站”命令，弹出“新建网站”对话框，在此可以迅速创建一个不同模板类型的网站项目，如图 2-11 所示。

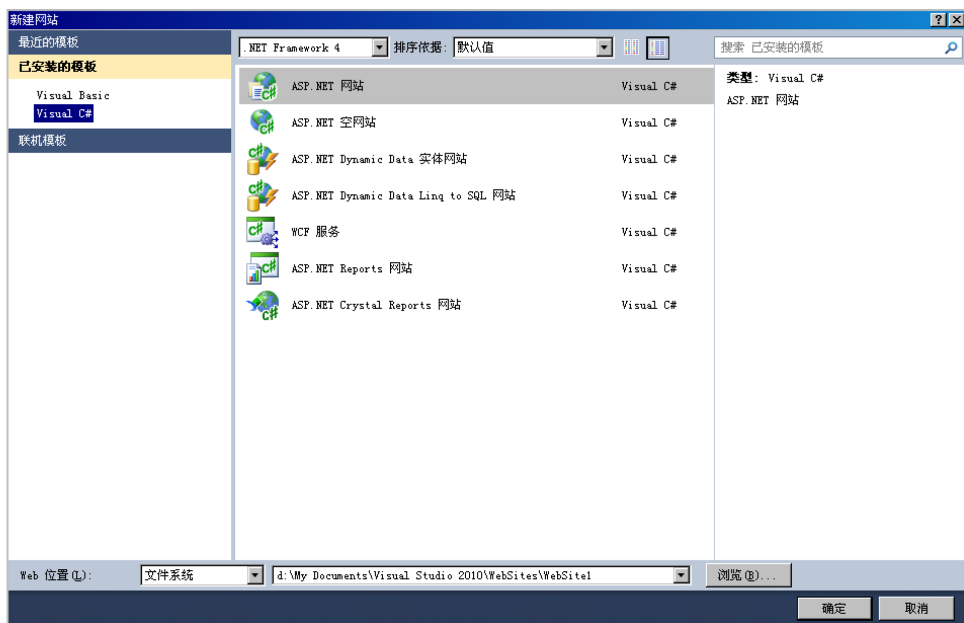


图 2-11　“新建网站”对话框

在菜单栏中依次单击“文件”|“新建”|“文件”命令，弹出“新建文件”对话框，在此可以迅速创建一个不同模板类型的文件，如图 2-12 所示。

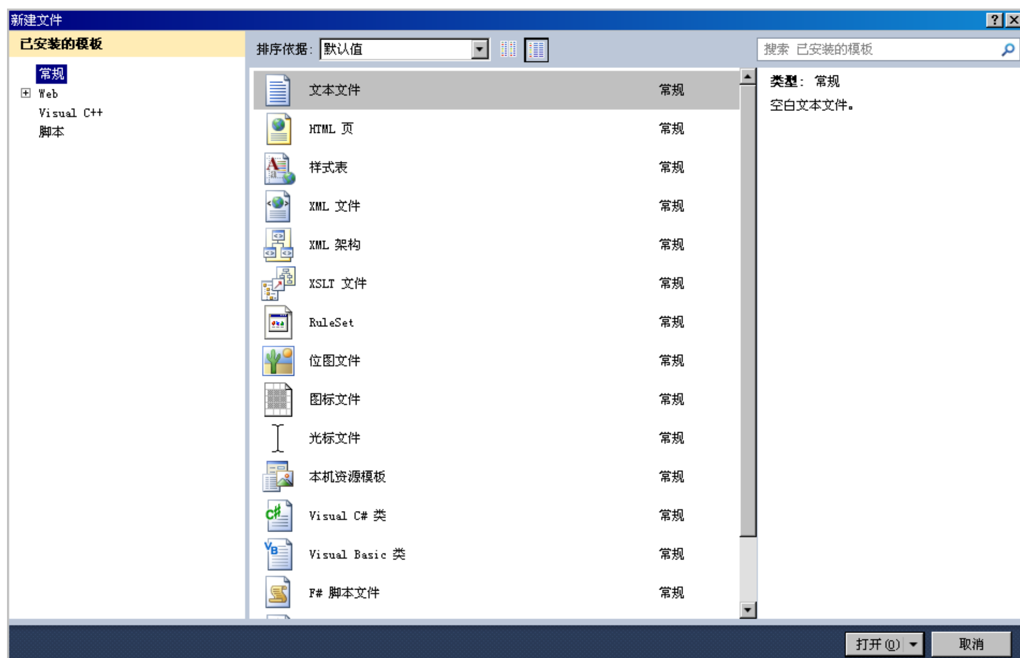


图 2-12 “新建文件”对话框

如果在 Visual Studio 2010 中创建一个项目，可以自动生成必需的代码。例如新建一个控制台项目后，将在项目文件内自动生成必需格式的代码，并且在右侧的“解决方案资源管理器”中显示自动生成的项目文件，如图 2-13 所示。

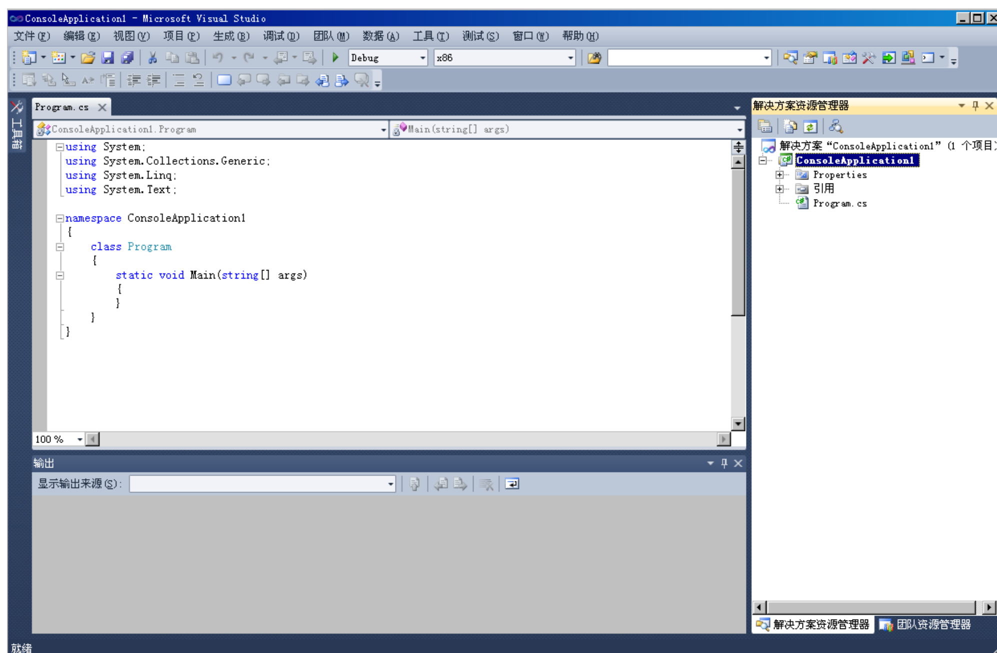


图 2-13 自动生成的代码和文件

2.2.3　解决方案和类视图

解决方案和类视图是 Visual Studio 2010 的重要组成工具，通过它们可以更加灵活地对项目进行控制和管理。下面将对 Visual Studio 2010 解决方案和类视图的基本知识进行简要介绍。

1．解决方案

当创建一个项目后，会在“解决方案资源管理器”中显示自动生成的项目文件。解决方案中包含一个或多个“项目”，每个项目都对应于软件中的一个模块。在资源管理器中，Visual Studio 2010 将同类的文件放在一个目录下，当单击这个目录后，会将对应目录的三个文件全部显示出来，如图 2-14 所示。例如，双击“引用”目录后，则将引用的程序集显示出来。

右键单击“解决方案资源管理器”中的每个节点，都将弹出一个上下文菜单，通过其中的菜单命令可以对节点对象进行操作。例如，右键单击项目名并依次选择“添加”｜“新建项”命令后，可以在项目内添加一个新的项目文件，如图 2-15 所示。

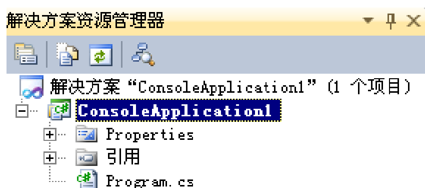


图 2-14　“引用”目录的程序集

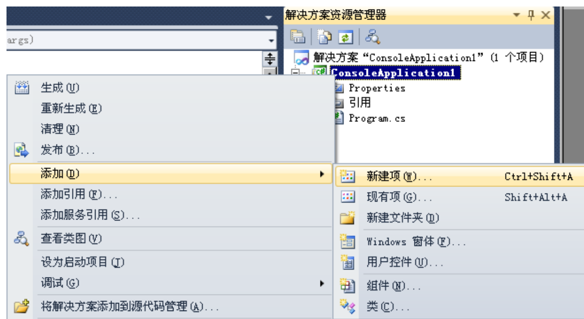


图 2-15　新建项目文件

2．类视图

上面介绍的“资源管理器”是以文件为对象的项目管理，而 C#是一种面向对象的编程语言，其基本的对象编程单位是类。为此，Visual Studio 2010 提供了类视图来进行对项目对象的管理。

在菜单栏中依次单击“视图”｜“类视图”命令，将在“类视图”中显示当前项目内的所有类对象，如图 2-16 所示。

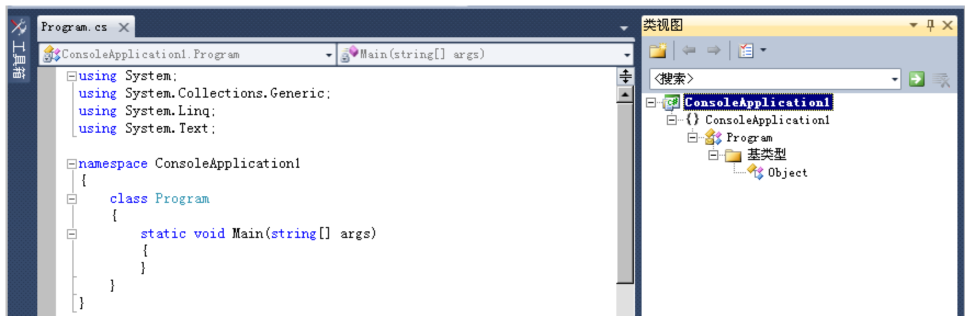


图 2-16　项目类视图

在图 2-16 中，上方显示项目的命名空间、基类和各种子类，具体说明如下。

- {}：表示命名空间。
- ：表示基类。
- ：表示普通类或子类。

在类视图中选中一个类的类型，然后单击右键将弹出一系列和类相关的操作命令，如图 2-17 所示。例如，选择“查看类图”可以查看这个类的关系图结构，并且可以在 Visual Studio 2010 的底部窗口查看类的详细信息，如图 2-18 所示。



图 2-17　类操作命令

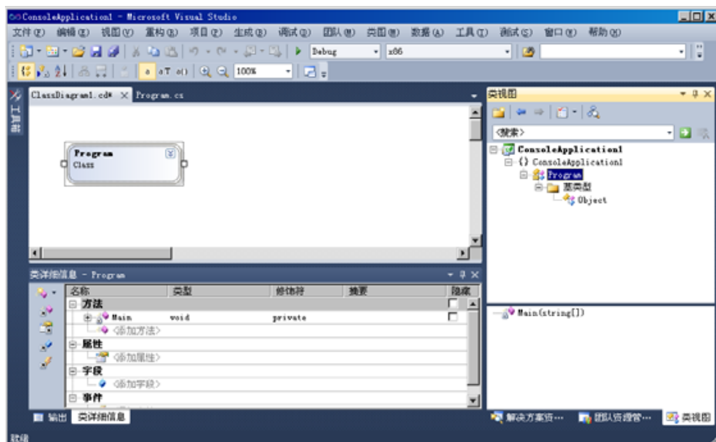


图 2-18　类关系结构和详细信息

2.2.4　文本编辑器

用户双击“解决方案资源管理器”的文件名，就能查看此文件的源代码。如果在 Visual Studio 2010 打开多个项目文件，则会在栏目内显示多个文件的文件名，如图 2-19 所示。



图 2-19　文件名栏

Visual Studio 2010 文本编辑器主要有如下几个特点。

（1）用不同的颜色显示不同的语法代码

在 Visual Studio 2010 文本编辑器中，使用蓝色显示 C#的关键字，用青色显示类名。

（2）代码段落格式自动调整

Visual Studio 2010 中文件源代码段落会自动缩进，这样可以加深代码的直观性。图 2-20 所示的就是源代码的段落缩进格式。

（3）语法提示

当用户使用文本编辑器进行代码编写时，编辑器能够根据用户的输入代码来提供对应的语法格式和关键字。例如，在图 2-20 所示的代码界面中输入字符“na”后，编辑器将自动弹出对应的提示字符，如图 2-21 所示。

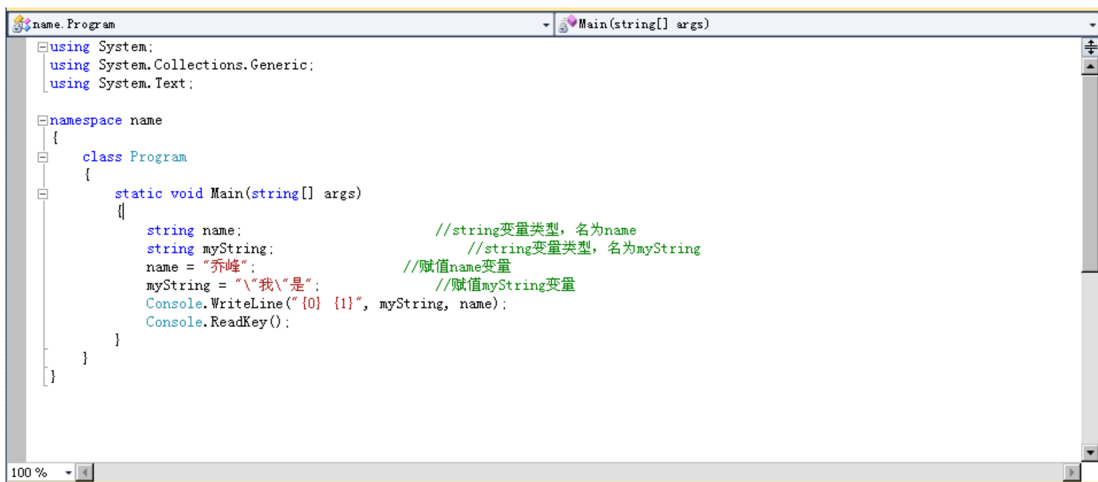


图 2-20　源代码段落缩进格式



图 2-21　语法提示效果

（4）行数显示

在 Visual Studio 2010 中会显示文件源代码的行数标记，这和 Dreamweaver 等工具一样，能够便于用户对程序的维护，迅速找到对应的代码所在。在初始安装 Visual Studio 2010 时，默认的是不显示代码行数。解决方法如下。

1）依次单击“工具栏”中的“工具”｜“选项”命令，弹出“选项”对话框，如图 2-22 所示。

2）在左侧的目录中依次单击“文本编辑器”｜“C#”｜“常规”，然后勾选右侧的“行号”复选框，如图 2-23 所示。

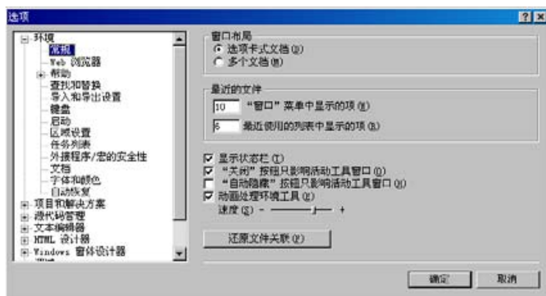


图 2-22　“选项”对话框

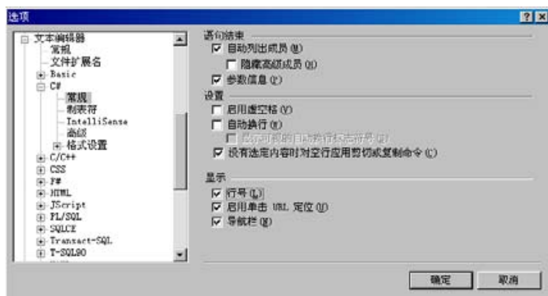


图 2-23　选中“行号”复选框

3）单击“确定”按钮后返回代码界面，此时文件中每行源代码前都将显示行号，如图 2-24 所示。

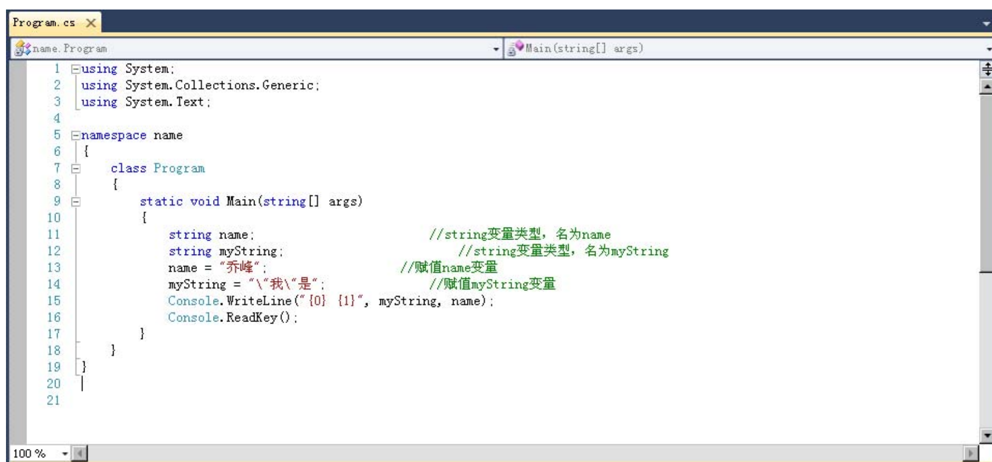


图 2–24　显示行号

2.2.5　生成与查错

依次单击 Visual Studio 2010 菜单栏中的“生成”｜“生成解决方案”命令，可以生成当前解决方案的所有项目。当使用“生成”命令时，不会编译已经生成过并且生成后没有被修改的文件。如果使用“重新生成”命令，则将重新生成所有的文件。

解决方案和项目有如下两种生成模式。

- ❑ 调试模式，即 Debug 模式。生成的代码中含有调试信息，可以进行源代码级的调试。
- ❑ 发布模式，即 Release 模式。生成的代码中不含有调试信息，不能进行源代码级的调试，但是运行速度较快。

开发人员可以依次单击菜单栏中的“生成”｜“配置管理器”命令，在弹出的“配置管理器”对话框中设置项目的生成模式，如图 2–25 所示。

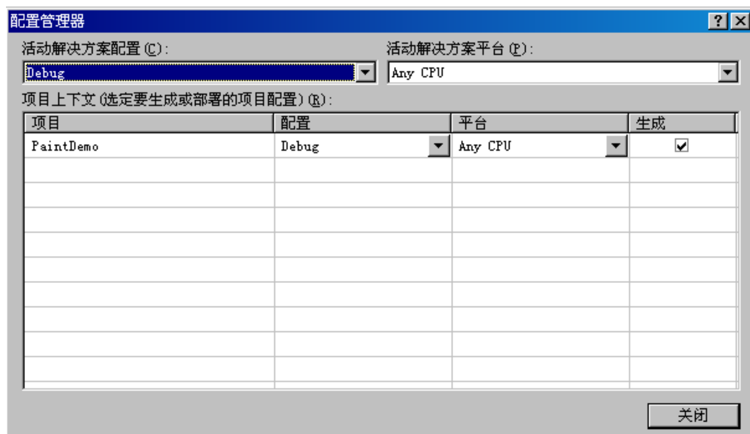


图 2–25　“配置管理器”对话框

如果项目中的代码出现错误，则不能成功生成，并在错误列表内输出错误提示，如

图 2-26 所示。

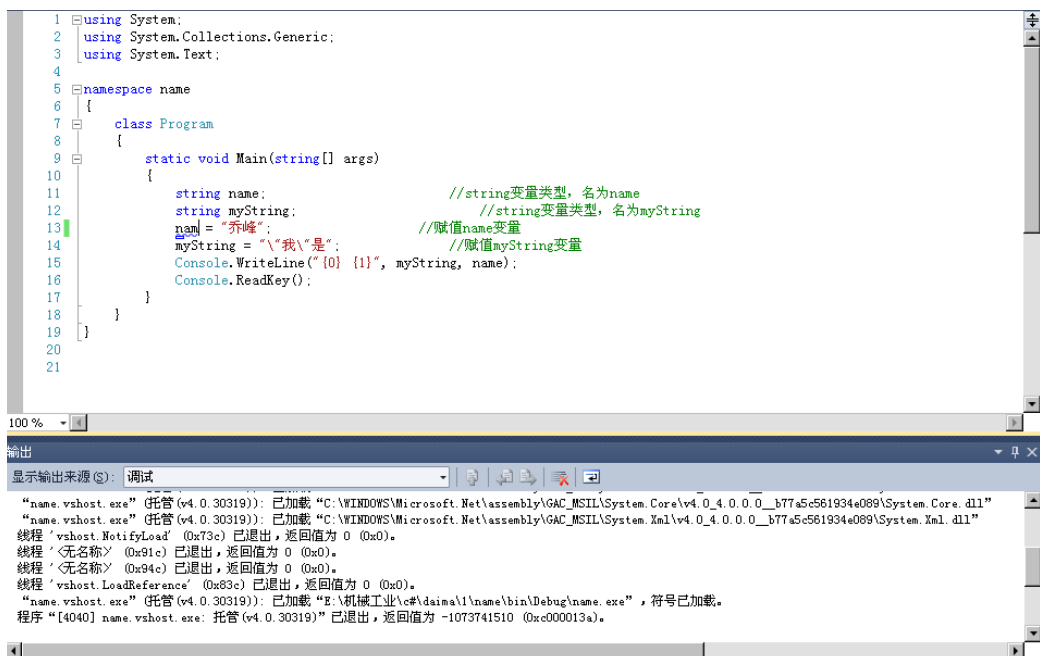


图 2-26　生成错误提示

Visual Studio 2010 能够进行查错处理，在"显示输出来源"下拉列表中选择"代码段输出"选项将显示出错信息详情，如图 2-27 所示。

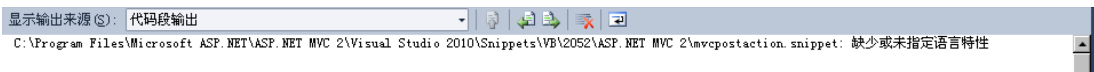


图 2-27　查错结果详情

如果将错误修改后则能正确生成，并在"输出"框内显示对应的处理结果，如图 2-28 所示。

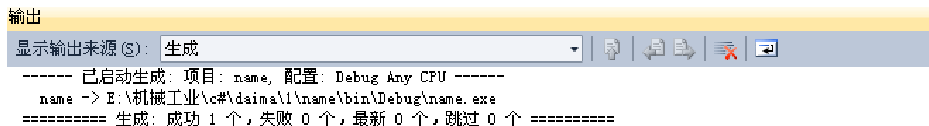


图 2-28　生成处理结果

2.3　疑难问题解析

本章详细介绍了搭建 C#开发环境的基本知识。本节将对本章中比较难以理解的问题进行讲解。

读者疑问：在安装 Windows 组件时，IIS 中的 Front Page 服务器扩展安装失败是怎么回事啊？提示道："将 XP Profession Service Pack 2 CD 插入选定的驱动器"。其实，我已经插入安装光盘了。难道需要修改注册表吗？怎么修改才能完成安装呢？

解答：笔者原来装 2010 的时候也是这样，但是笔者当时没光盘。按理说，把光盘插进去应该可以通过的。笔者当时是这样解决的，出现"将 XP Profession Service Pack 2 CD 插入选定的驱动器"提示，是因为 Windows 的系统文件保护不让你通过，那只好关闭文件保护了，关闭方法如下。

1）运行 gpedit.msc 打开组策略。

2）依次展开"计算机配置"|"管理模板"|"系统"|"Windows 文件保护"。

3）找到"设置文件保护"，双击并修改为"已禁用"，重新启动系统，然后安装 2010 就可以了。

上面的方法最可行，但治标不治本。还有一种方法，就是彻底修复 Windows 文件。插入系统安装光盘，并运行 sfc/scannow 命令检测并修复可能受损坏和被更改的系统文件。这样，提示就不会再出现了。建议先试第一种方法再试第二种。

读者疑问：在安装 Visual Studio 2010 时，因为版本问题，不能安装在 Windows XP 系统，该怎样解决？

解答：在 Windows XP SP2 上安装 Visual Studio 2010，具体的做法相当简单。在命令行浏览到安装目录，输入:

```
setup.exe /NO_BSLN_CHECK
```

就可自动绕过安装的操作系统验证了。



职场点拨——怎样学好 C#

学习程序开发之路是充满挑战之路，枯燥的代码和烦琐的调试有时会使你感到枯燥无味；但同时也是充满乐趣之路，每一个功能的调试成功都会使你充满自豪和成就感。学习 C#也是如此，学习之路上肯定会充满挑战也充满乐趣。作为一名初学者，怎样才能学好 C#呢？下面给出几点建议。

（1）培养兴趣

兴趣是学习任何知识的动力。在现实中，往往我们会对喜欢的事情充满热情，也乐于耗费精力。对于编程来说，只要喜欢感受那种调试成功的喜悦，就说明你已经对编程产生了兴趣。另外，调试成功的喜悦会让你更加喜欢编程，更能带来成就感。

（2）多看代码，多实践

当有一定的语法基础以后，一定要多看别人的代码，看的目的是掌握程序的结构和流程，看完之后需要自己亲自实践。程序开发讲究精细，哪怕是一个标点的错误都不会调试成功。有人说学习编程的秘诀是编程、编程、再编程，练习、练习、再练习，这就充分说明了实践的重要性。

编程讲究实践，光说不练是不行的，要遵循实践检验理论的原则。在刚开始学的时候

可以练习书上的习题，如果遇到不明白的地方，最好亲自编写一个小程序进行验证，这样能给自己留下深刻的印象。在自己动手的过程中要不断纠正自己不好的编程习惯和错误认识。在有一定的基础以后，可以尝试编一些小游戏，由几个网页构成的简单站点。基础很扎实的时候，可以编一些大型 Web 系统或桌面程序。网上源代码资源丰富，可以获取后分析这些代码。

（3）脚踏实地，稳扎稳打

急功近利是贬义词，欲速则不达也是教育人时的常用词语，学编程时也要时刻注意。很多初学者刚学会了基本语法知识，调试成功了几段代码，就迫不及待地大声宣布“我精通C#了”。但是当遇到问题之后才发现，自己学到的仅仅是九牛一毛。前人的忠告是——基础一定要打牢，基本语法是编程的根本。这样才能确保以后工作项目的顺利进行。

第 3 章　变量与常量

C#语言本身强大的语法特性决定了它强大的功能。读者在学习过程中，应该首先掌握C#语法等基本知识，才能真正学好 C#这种语言。所以从本章开始，将引导读者真正进入 C#世界，从其最基本的语法内容开始，进行循序渐进的学习。变量和常量是任何一种高级编程语言的必需要素。本章将简要介绍 C#的变量和常量的基本知识，并通过简单的实例来介绍其具体使用流程。通过本章能学到如下知识。

- C#的基本语法。
- 变量、常量。
- 类型转换。
- 基本.NET 框架类。
- 职场点拨——和客户相处之道。

从今天算起，小菜步入职场正好满一个月，职位是产品部客户服务专员。

2006 年 X 月 X 日，多云

明天是周一了，已经提前约好了上午 10：00 在公司会见客户张总。这是我工作后第一次独自接待客户，感觉有点紧张，我不知该怎么办，甚至都不知道该穿什么样的衣服。



一问一答

Wisdom：看你焦急的样子，在为什么事情发愁呢？

小菜：明天客户就要来了，这是我第一次独自接待客户，心里很没底！

Wisdom：不用担心，客户其实很好相处的，只要你明确对方是什么类型的客户，此行有什么目的，你在交流时就会胸有成竹。本章最后的“和客户相处之道”，相信会对你有所帮助。

小菜：嗯，言归正传，本章将要学的变量和常量很重要吗？

Wisdom：就像客户有不同的类型和目的一样，C#中的数据也有不用的类型，但从最基本的要素上分为变量和常量。

3.1　C#的基本语法

1．C#的特性

因为 C#是从 C 和 C++进化而来的，所以从外观和语法定义上看，C#和两者有着很多相

似之处。在使用 C#进行代码编写的过程中，必须遵循它本身的独有特性，即基本的语法结构。C#的代码语句具有如下 4 个特性。

（1）字符过滤性

和其他常用语言的编译器不同，无论代码中是否含有空格、换行符或制表符，C#都不会考虑它们。这样，程序员在编写代码时会有很大的自由度，不会因为疏忽加入空白字符而造成程序的错误。

（2）语句结构

C#程序代码是由一系列的语句构成的，并且每个语句都必须以分号“;”结束。因为 C#中的空格和换行等字符被忽略，所以可以在同行代码中放置多个处理语句。

（3）代码块

因为 C#是面向对象的语言，所以其代码结构十分严谨和清晰。同功能的 C#代码语句构成了独立的代码块，通过这些代码块可以使整个代码的结构更加清晰。所以说，C#代码块是整个 C#代码的核心。

（4）严格区分字母大小写

因为在 C#语言中字母的大小写代表不同的含义，所以在编写代码时，必须注意每个字母的大小写格式，避免因大小写差异而出现名称错误的问题。

（5）C#的代码块以“{”开始，以“}”结束。

根据上述特性，总结出 C#代码块的基本语法结构如下：

```
语句 1; {
    语句 2;
    语句 3;
    ……;
}
{
    语句 m;
          {
            语句 n;
            ……;
           }
}
```

在上述格式中，使用了缩进格式和非过滤处理。上述做法的目的是使整个 C#代码变得更加清晰，提高代码的可读性。在此向读者提出如下两点建议。

1）独立语句独立代码行。虽然 C#允许在同一行内放置多个 C#语句，但为了提高代码的可读性，建议将每个语句放置在独立的代码行中，即每行代码都以分号“;”结束。

2）代码缩进处理。对程序内的每个代码块都要遵循缩进原则，使各代码块在整个程序中以更加清晰的效果展现出来。在使用 Visual Studio 进行 C#开发时，Visual Studio 能够自动实现代码缩进。

2．C#的注释方法

C#程序中的必要构成元素是注释。通过注释可以帮助程序员和用户快速了解当前语句的

功能。特别是在大型应用程序中，因为整个项目内的代码块繁多，所以加入合理的注释必不可少。在 C#中加入注释的方法有如下两种。

（1）两端放置

两端放置是指在程序的开头和结尾放置，具体格式是在开头插入“/*”，在结尾插入“*/”，在两者之间输入注释的内容。例如下面的代码：

```
/* 代码开始了 */
static void Main(string[] args)
      {
         int myInteger;
         string myString;
```

（2）单“//”标记

单“//”标记和上面的两端放置不同，其最大特点是以“//”为注释的开始，在注释内容编写完毕后，只需注释内容和开头“//”在同行即可。例如下面的代码：

```
//代码开始了
static void Main(string[] args)
      {
         int myInteger;
          string myString;
```

但是下面的代码是错误的：

```
//代码开始了
还是注释
static void Main(string[] args)
      {
         int myInteger;
          string myString;
```

下面通过一段简单的 C#代码，向读者说明 C#程序的基本语法结构。实例的具体实现流程如下。

1）打开 Visual Studio 2010，依次单击“新建”｜“项目”选项，弹出“新建项目”对话框，如图 3-1 所示。

2）选择“控制台应用程序”选项，设置“名称”为 mm，然后单击“确定”按钮进入代码界面。

在 Visual Studio 2010 中自动创建的 C#格式代码如下：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace mm                            /* 名称 */
{
```

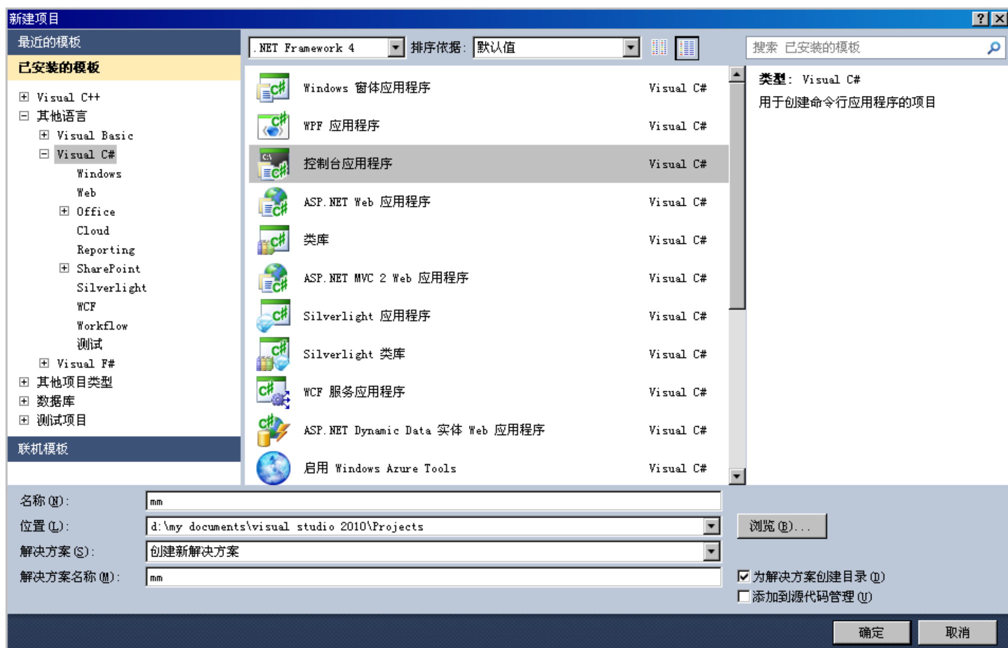


图 3-1 “新建项目”对话框

```
        class Program                      //类
        {
            static void Main(string[] args)
            {
            }
        }
    }
```

在上述代码中，可以先不必考虑各代码的具体含义，而只看它们的语法结构格式。经过上述实例代码，可以明确 C#语句的各个主要构成元素。

3.2 变量

在 C#语言中，变量表示内存地址的名称。C#变量包括三个主要元素，分别是名称、类型和值。上述元素的具体说明如下。

- ❑ 变量名：变量在程序代码中的标识。
- ❑ 变量类型：决定其所代表的内存大小和类型。
- ❑ 变量值：变量所代表内存块中的数据。

3.2.1 C#类型

因为 C#支持 Visual Studio 2010 框架定义的类，所以 C#的变量类型是用类来定义的，即所有的类型都是类。C#类型的具体说明如表 3-1 所示。

表 3-1　C#类型信息

类　　型		描　　述
值类型	简单类型	符号整形：sbyte，shote，int，long
		无符号整形：byte，ushort，uint，ulong
		unicode 字符：char
		浮点型：float，double
		精度小数：decimal
		布尔型：bool
	枚举类型	枚举定义：enum name{}
	结构类型	结构定义：stroct name{}
引用类型	类类型	最终基类：object
		字符串：string
		定义类型：class name
	接口类型	接口定义：interface
	数组类型	数组定义：int[]
	委托类型	委托定义：delegate name

由表 3-1 所示信息可以看出，C#变量的常用类型有值类型和引用类型两大类。

1．引用类型

引用类型是 C#的主要类型，在引用变量中保存的是对象的内存地址。引用类型具有如下五个特点。

1）需要在委托中为引用类型变量分配内存。

2）需要使用 new 运算符创建饮用类型的变量，并返回创建对象的地址。

3）引用类型变量是由垃圾回收机制来处理的。

4）多个引用类型变量都可以引用同一对象，对一个变量的操作会影响到另一个变量所引用的同一对象。

5）引用类型变量在被赋值前的值都是 null。

C#中所有被称为类的变量类型都是引用类型，包括类、接口、数组和委托，具体说明如下。

- 类类型：能够定义包含数据成员、函数成员和嵌套类型的数据结构。其中，数据成员包括常量和字段；函数成员包括方法、属性和事件等。
- 接口：能够定义一个协议，实现某接口的类或结构必须遵循该接口定义的协议。
- 数组：一种数据结构，包含可通过计算索引访问的任意变量。
- 委托：一种数据结构，能够引用一个或多个方法。

2．值类型

如果程序中只有引用类型，则会很容易影响整个程序的性能，通过值类型能够很好地解决这个问题。值类型是组成应用程序的最为常见的类型之一，能够存储应用数值。例如，通过一个名为 mm 的变量存储数值 100，在应用时只需调用变量名 mm，即可调用数值 100。

值类型的主要特点如下。

- 值类型变量被保存在堆栈中。

- 在访问值类型变量时，一般直接访问其实例名。
- 每个值类型变量都有本身的副本，所以对一个值类型变量的操作不会影响到其他变量。
- 在复制值类型变量时，复制的是变量的值，而不是变量的地址。
- 值类型变量的值不能是 null。

值类型从 System.ValueType 类中继承类，包括结构、枚举和大多数的基本类型。具体说明如下：

- 结构类型，能够声明常量、字段、方法和属性等。
- 枚举类型，具有命名常量的独特类型，每个枚举类型都有一个基础的类型，是通过枚举来声明的。

3．基本类型

基本类型是编译器直接支持的类型。使用关键字来命名基本类型，它是构造其他类型的基础。其中值类型的基本类型通常被称为简单类型，例如下面的代码声明了 int 类型变量：

```
int mm=123;
```

下面将详细介绍常用的基本类型。

（1）整形

在 C#中定义了八种整形，具体说明如表 3-2 所示。

表 3-2　C#整型信息

类　　型	值 的 范 围
sbyte	–128～127 的整数
byte	0～255 的整数
short	–32768～32767 的整数
ushort	0～65535 的整数
int	–2147483648～2147483647 的整数
uint	0～4294967295 的整数
long	–9223372036854775808～–2147483647 的整数
ulong	0～18446744073709551615 的整数



注意

某变量前的字符“u”表示不能在此变量中存储负值。

（2）浮点型

浮点型包括 float 型和 double 型两种，具体说明如表 3-3 所示。

表 3-3　C#浮点型信息

类　　型	允许值的范围
float	32 位浮点数，精度是 7 位
double	64 位浮点数，精度是 15～16 位

（3）布尔型

布尔型有两个取值，分别是 true 和 false，即代表“是”和“否”的含义。

（4）字符型

字符型的取值和 Unicode 的字符集相对应，通过字符型数据可以表示世界上所有语言的字符。字符型文本一般用一对单引号来标识，例如'MM'和'NN'。

使用字符型的转义字符可以表示一些特殊字符，常用的转义字符如表 3-4 所示。

表 3-4　C#转义字符列表

转 义 字 符	描　　述
\'	转义单引号
\"	转义双引号
\\	转义反斜杠
\0	转义空字符
\a	转义感叹号
\b	转义退格
\f	转义换页
\n	转义新的行
\r	转义回车
\t	转义水平制表符
\v	转义垂直制表符
\x	后面接两个二进制数字，表示一个 ASCII 字符
\u	后面接四个二进制数字，表示一个 ASCII 字符

（5）decimal 型

decimal 型是一种高精度的、128 位的数据类型，常用于金融和货币计算项目。decimal 型表示 28 或 29 个有效数字，取值范围为±1×10～7×10。

（6）string 型

string 型用来表示字符串，常用于文本字符的代替，是字符型对象（char）的连续集合。当创建 string 型的字符串值后，就不能再修改，除非重新赋值。string 型的变量赋值需要用双引号括起来，例如下面的代码：

```
string mm="好";
```

另外，string 的变量可以使用“+”连接变量字符串。例如，下面的代码将输出“你好，机械工业出版社”。

```
string mm="你好";
string nn=",";
string zz="机械工业出版社";
string ff="! ";
string jieguo=mm+nn+zz+ff;
Console.WriteLine(jieguo);
```

另外，也可以使用“+”连接不同数据类型的字符串。例如，下面的代码将输出“Hi，123！”。

```
string mm="Hi";
string nn=",";
int zz=123;
string ff="! ";
string jieguo=mm+nn+zz+ff;
Console.WriteLine(jieguo);
```

（7）object 型

object 型是 C#最基础的类型，它可以表示任何类型的值。

实例 1：使用 C#变量输出“"我"是帅哥”

下面通过一个具体的实例来说明 C#变量的使用。本实例保存在“光盘:\daima\3”文件夹内，项目名为 name。具体实现流程如下。

1）打开 Visual Studio 2010，依次单击“新建”｜“项目”选项，弹出“新建项目”对话框，如图 3-2 所示。

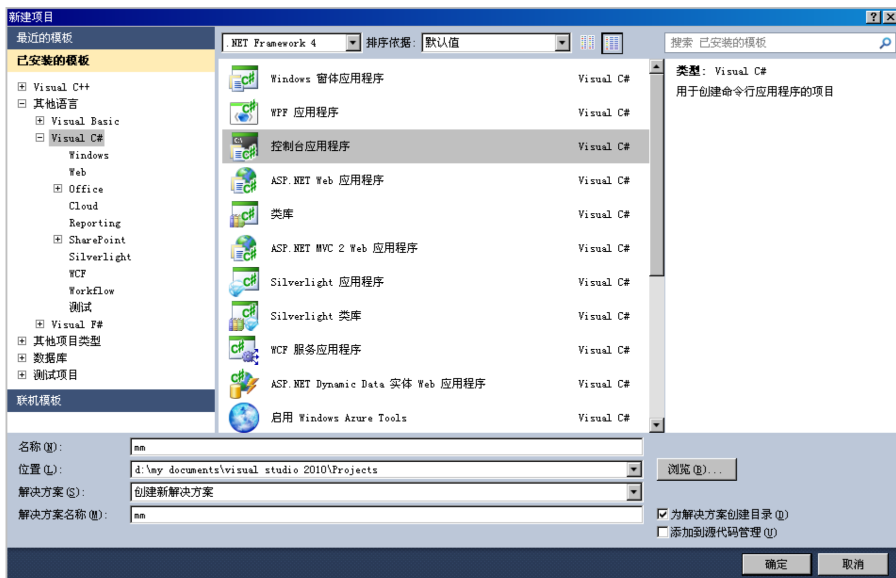


图 3-2　“新建项目”对话框

2）选择“控制台应用程序”选项，设置“名称”为 name，然后单击“确定”按钮进入代码界面。

3）输入对应的代码，项目文件 Program.cs 整理后的主要代码如下：

```
namespace name
{
    class Program
    {
        static void Main(string[] args)
```

```
            {
                string name;                    //string 变量类型，名为 name
                string myString;                //string 变量类型，名为 myString
                name = "帅哥";                  //赋值 name 变量
                myString = "\"我\"是";          //赋值 myString 变量
                Console.WriteLine("{0} {1}", myString, name);
                Console.ReadKey();
            }
        }
    }
```

在上述实例代码中，设置了两个变量 name 和 myString，具体说明如下。

- 变量 name：类型为 string，并设置其值为“帅哥”。
- 变量 myString：类型为 string，并设置其值为“"我"是”。其中字符“\"”的功能是转义双引号。

实例的最终结果是输出两变量的连接值，具体如图 3-3 所示。



图 3-3　实例执行结果

多学一招

在上述实例中，使用了 Visual Studio 命令进行编译。除了上述编译方式外，读者也可以使用 SDK 命令进行编译，并且可以同时使用编译命令对多个.cs 文件进行编译。例如，某个项目中有 123.cs 和 456.cs 两个文件，则可以通过“csc /out:name.exe 123.cs 456.cs”将其编译为一个可执行文件“name.cs”。上述实例只是使用了“/out”编译参数，其他编译参数的使用方式与其相似。

3.2.2　给变量命名

在 C#中不能给变量任意命名，必须遵循如下两条原则。

1）变量名的第一个字符必须是字母、下画线“-”或@。

2）第一个字符后的字符可以是字母、下画线或数字。

另外，还需要特别注意 C#编译器中的关键字，例如关键字 using。如果错误地使用了编译器中的关键字，则程序将会出现编译错误。例如下面的变量名都是正确的：

```
aaaaaa
@aaaaaaa
_aaaaaa
```

而下面的变量名是不正确的：

```
6666aaa
```

```
aa-bb
namespace
```

因为 C#是区分大小写的，所以在命名过程中要注意区分大小写。为了使编写的程序结构更加清晰明了，建议读者根据变量的作用来命名，并区分大小写字母。例如，存储名称的变量命名为 Name，存储用户名的变量命名为 UserName。

3.2.3　变量的声明和赋值

C#中声明变量的具体格式如下：

```
类型 变量名;
```

例如，下面的代码声明了一个名为 Name 的 string 变量。

```
string Name;
```

在 C#中可以同时声明多个变量，各变量之间需要用逗号“,”隔开。例如，下面的代码同时声明了 aa、bb 和 cc 三个变量。

```
string aa,bb,cc;
```

在声明多个变量的同时，可以对各变量进行赋值，例如下面的代码：

```
string aa= "我",bb= "是",cc= "谁";
```

因为 C#是类型安全的语言，所以当给一个变量赋值时，值的类型必须满足如下条件之一。

- 与变量的类型相同。
- 是一个 C#将执行赋值转换的类型。
- 是一个可以显示转换为正确类型的类型。

由前面的知识了解到，C#变量在命名时必须遵循特定的规则，并且 C#变量在使用前必须进行初始化处理。例如，下面的首行代码进行了初始化，而第二行代码进行了赋值。

```
int age;
age=26;
```

而下面的代码将会出现编译错误。

```
namespace name
{
    class Program
    {
        static void Main(string[] args)
        {
            string name;
            string myString;
            name = myString
            Console.WriteLine("{0} {1}", myString, name);
            Console.ReadKey();
```

```
            }
        }
    }
```

3.3 常量

通常将值是固定不变的变量称为常量，常量的值在编译时就已经确定下来。在 C#中的常量类型只能是下列类型中的一种：sbyte、byte、ushort、short、int、uint、ulong、long、char、float、double、decimal、bool、string 或枚举。

通常将 C#常量分为两种，分别是文本常量和符号常量。具体说明如下。

1）文本常量，是输入到程序中的值，例如“12”和“Mr.王”等；符号常量和文本常量类似，是代表内存地址的名称，在定义后就不能修改了。

2）符号常量，其声明方法和变量的声明方法类似，唯一的区别是常量在声明前必须使用修饰关键字 const 开头，并且常量在定义时被初始化。常量一旦被定义后，在常量的作用域内其本身的名字和初始化值是等价的。

符号常量的命名规则和变量的命名规则相同，但是符号常量名的第一个字母最好是大写字母，并且在同一个作用域内，所有的变量名和常量不能重名。例如下面的常量命名是正确的。

```
const double mm=3.14;
const string nn=LiuDeHua;
```

而下面的常量命名是错误的。

```
const double mm;
string nn=LiuDeHua;
```

3.4 类型转换

在计算机的类型中，所有的数据都是由 0 和 1 构成的一系列位。C#中最简单的数据类型是 char 类型，它可以用一个数字来表示 Unicode 字符集中的一个字符。在默认情况下，不同类型的变量使用不同的模式来表示数据。也就是说，当变量值经过移位处理后，所得到的结果将会不同，这样类型转换就提上了议程。C#中的类型转换有隐式转换和显式转换两种，下面将分别介绍。

3.4.1 隐式转换

隐式转换指不需要特别声明即可在所有情况下进行的转换，这是系统的默认转换方式。在进行隐式转换时，编译器不需要进行检查就能实现安全的转换处理。C#的隐式转换一般不会失败，也不会导致信息丢失。例如，下面的代码从 int 类型隐式转换成了 long 类型。

```
int mm=20;
```

```
    long nn=mm;
```

在 C#中的简单类型有许多隐式转换，但是其中的 bool 和 string 是没有隐式转换的。编译器可以隐式执行的数值转换类型如表 3-5 所示。

表 3-5　C#可隐式转换的数值类型列表

类　　型	可转换成的类型
byte	short、ushort、uint、int、ulong、long、float、double、decimal
sbyte	short、int、long、float、double、decimal
short	int、long、double、decimal
ushort	uint、int、ulong、long、float、double、decimal
int	long、float、double、decimal
uint	ulong、long、float、double、decimal
long	float、double、decimal
ulong	float、double、decimal
float	double
char	ushort、uint、int、ulong、long、float、double、decimal



不存在 char 类型的隐式转换，所以其他整形值不会自动转换为 char 类型。另外，读者不需要强记表 3-5 的内容，只需牢记各类型的取值范围即可。因为对于任何类型 A，只要其取值范围被完全包含在类型 B 的取值范围内，就可以隐式地将类型 A 转换为类型 B。

实例 2：使用隐式转换

下面通过一个具体的实例来说明 C#隐式转换的具体效果。本实例保存在“光盘:\daima\3”文件夹内，项目名为 Yinshi。具体实现流程如下：

1）打开 Visual Studio 2010，依次单击“新建”｜“项目”选项，弹出“新建项目”对话框，如图 3-4 所示。

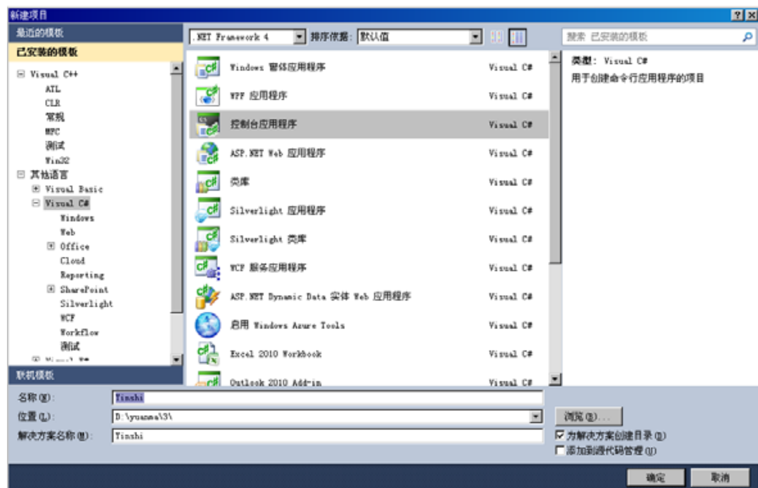


图 3-4　“新建项目”对话框

2）选择“控制台应用程序”选项，设置“名称”为 YinshiZhuanhuan，然后单击“确定”按钮进入代码界面。

3）输入对应的代码，项目文件 Program.cs 整理后的主要代码如下：

```
namespace Yinshi
{
    class Yinshi
    {
        public static void Main()
        {
            byte aa = 100;
            int bb = aa;
            long cc = bb;
            double dd = cc;
            Console.WriteLine("mm = {0}", aa);
            Console.WriteLine("nn = {0}", bb);
            Console.WriteLine("zz = {0}", cc);
            Console.WriteLine("ff = {0}", dd);
            Console.ReadKey();
        }
    }
}
```

在上述实例代码中，设置了四个不同类型的变量，具体说明如下。

- 变量 aa：类型为 byte，并设置其值为“100”。
- 变量 bb：类型为 int，并将变量 mm 隐式转换为 int 类型。
- 变量 cc：类型为 long，并将变量 nn 隐式转换为 zz 类型。
- 变量 dd：类型为 double，并将变量 zz 隐式转换为 ff 类型。

实例的最终输出结果是四个变量的值相同，实现了隐式转换，结果如图 3-5 所示。

图 3-5　实例执行结果

3.4.2　显式转换

显式转换是一种强制性的转换方式，在使用显式转换时，必须在代码中明确声明要转换的类型。为此需要编写特定的额外代码，并且代码的格式会随着转换方式的不同而不同。

虽然 C#允许变量进行显式转换，但需要注意下面两点。

1）隐式转换是显式转换的一种特例，所以允许把隐式转换的转换书写成显式转换格式。

2）显式转换不安全，因为不同类型的变量取值范围是不同的，所以如果强制执行显式转换，可能会造成数据丢失。

C#显式转换的语法格式如下：

```
类型 变量名=（类型）变量名
```

下面通过两段代码来看显式转换和隐式转换的区别。

隐式转换的代码如下：

```
int mm=20;
long nn=mm;                                    //隐式转换
```

显式转换的代码如下：

```
int mm=20;
long nn=(long)mm;                              //显式转换
```

从一个数值的类型向另一个数值类型进行转换的过程被称为显式数值转换。因为显式转换包括隐式转换和显式的数值转换，虽然在转换过程中会出现数据丢失，但也总是能强制转换表达式从任何数值类型转换为任何其他类型。

编译器可以显式执行的数值转换类型如表 3-6 所示。

表 3-6　C#可显式转换的数值类型列表

类　　型	可转换成的类型
byte	sbyte、char
sbyte	byte、ushort、uint、ulong、char
short	sbyte、byte、ushort、uint、ulong、long、char
ushort	sbyte、byte、short、char
int	sbyte、byte、short、ushort、uint、ulong、char
uint	sbyte、byte、short、ushort、int、char
long	sbyte、byte、short、ushort、uint、int、ulong、char
ulong	sbyte、byte、short、ushort、uint、int、long、char
float	sbyte、byte、short、ushort、uint、int、ulong、long、char、float
char	sbyte、byte、short
decimal	sbyte、byte、short、ushort、uint、int、ulong、long、char、float、double
double	sbyte、byte、short、ushort、uint、int、ulong、long、char、float、decimal

实例 3：使用 C#显式转换

下面通过一个具体的实例来说明使用 C#显式转换的过程。本实例保存在“光盘:\daima\3”文件夹内，项目名为 Xianshizhuanhuan。实例文件的主要实现代码如下：

```
namespace Xianshi
{
    class Program
    {
        public static void Main()
        {
            double mm = 10000;
            long nn = (long)mm;
            int zz = (int)nn;
            byte ff = (byte)zz;
            Console.WriteLine("mm = {0}", mm);
```

```
                Console.WriteLine("nn = {0}", nn);
                Console.WriteLine("zz = {0}", zz);
                Console.WriteLine("ff = {0}", ff);
                Console.ReadKey();
            }
        }
    }
```

在上述实例代码中，首先定义了变量 mm 的数值类型为 double，且初始值为 10000；然后对 mm 值进行了显式转换，具体流程如下。

1）将 mm 转换为 long 数值类型，并赋值给变量 nn。

2）将 nn 转换为 int 数值类型，并赋值给变量 zz。

3）将 zz 转换为 byte 数值类型，并赋值给变量 ff。

经过上述转换处理，程序执行后的显示结果如图 3-6 所示。

```
file:///F:/机械视频/C#/daima/
mm = 10000
nn = 10000
zz = 10000
ff = 16
```

图 3-6　实例执行结果

3.4.3　装箱与拆箱

从实质上讲，C#中的值类型和引用类型是同源的，所以不但可以在值类型之间和引用类型之间进行转换，也可以在值类型和引用类型之间进行转换。但是由于两者使用的内存类型不同，使它们之间的转换变得比较复杂。

在 C#中，将值类型转换为引用类型的过程叫做装箱，将引用类型转换为值类型的过程叫做拆箱。装箱和拆箱在值类型和引用类型之间架起了一座沟通的桥梁，是整个 C#类型系统的核心知识。最终使任何值类型的值都可以转换为 object 类型值，object 类型值也可以转换为任何类型的值，并把类型值作为一个对象来处理。

1．装箱

装箱允许将值类型转换为引用类型，能够实现下述转换操作。

1）从任何值类型到 object 类型的转换。

2）从任何值类型到 System.ValueType 类型的转换。

3）从任何值类型到值类型的接口转换。

4）从任何枚举类型到 System.Enum 类型的转换。

将一个值类型装箱为一个引用类型的流程如下。

1）在托管堆中创建一个新的对象实例，并分配对应的内存。

2）将值类型变量值复制到对象实例中。

3）将对象实例地址压入堆栈中，并指向一个引用类型。

看下面的代码：

```
namespace ZhuangxiangChuli{
    class Program
    {
        public static void Main()
    {
        int mm = 100;                  // 定义值类型变量
        object nn = mm;                // 将值类型变量值装箱到引用类型对象
```

```
            Console.WriteLine("值为{0}，装箱对象为{1}", mm, nn);
            mm = 200;                    // 改变值
            Console.WriteLine("值为{0}，装箱对象为{1}", mm, nn);
            Console.ReadKey();
        }
    }
}
```

在上述代码中，首先定义了变量 mm 的数值类型为 int，且初始值为 100；然后对 mm 的值进行装箱，处理为对象 nn，具体如下。

- ❑ 将变量 mm 的值 100 进行装箱，转换为对象 nn。
- ❑ 将变量 mm 的值修改为 200 然后进行装箱，转换为对象 nn。

上述实例代码保存在“光盘:\3”文件夹内，名为 ZhuangxiangChuli。经过上述两次装箱处理，执行后的结果如图 3-7 所示。

从图 3-7 所示的结果可以看出，值类型的变量值被复制到装箱得到的对象中。装箱后如果改变值类型的值，并不会影响到装箱对象的值。

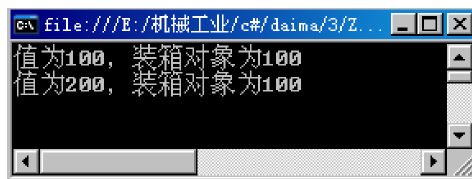


图 3-7　实例执行结果

2．拆箱

拆箱允许将引用类型转换为值类型，能够实现下述转换操作。

- ❑ 从 object 类型到任何值类型的转换。
- ❑ 从 System.ValueType 类型到任何值类型的转换。
- ❑ 从任何接口类型到对应的任何值类型的转换。
- ❑ 从 System.Enum 类型到任何枚举类型的转换。

将一个值类型装箱为一个引用类型的流程如下：

1）检查该对象实例是否是某个给定值类型装箱后的值。

2）如果是，则从实例中复制这个值。

3）为类型变量赋这个值。

实例 4：使用 C#拆箱操作

下面通过一个具体的实例来说明 C# 拆箱操作的实现过程。本实例保存在“光盘:\daima\3”文件夹内，项目名为 ChaixiangChuli。实例文件的主要代码如下：

```
namespace Chaixiang {
    class Program
    {
        public static void Main()
        {
            int aa =200;          // 定义一个值类型变量
            object bb = aa;       // 将值类型变量的值装箱到一个引用类型对象中
            Console.WriteLine("装箱处理：值为{0}，装箱对象为{1}", aa, bb);
            int cc = (int)bb;    // 取消装箱
            Console.WriteLine("拆箱处理：装箱对象为{0}，值为{1}", bb, cc);
```

```
                    Console.ReadKey();
                }
            }
        }
```

在上述实例代码中，首先定义了变量 mm 的数值类型为 int，且初始值为 200；然后对 aa 的值进行装箱，处理为对象 bb，具体流程如下。

1）将变量 aa 的值 200 进行装箱，转换为对象 bb。

2）将变量 bb 的值进行拆箱处理，转换为变量 cc。

经过上述两次处理，程序执行后的显示结果如图 3-8 所示。


多学一招

在拆箱处理过程中，必须保证处理变量的类型一致，否则将会出现异常。例如将上述实例中的拆箱处理代码进行如下修改：

```
        double cc = (double)bb;
```

修改后的执行结果如图 3-9 所示。

```
装箱处理：值为200，装箱对象为200
拆箱处理：装箱对象为200，值为200
```

图 3-8　实例执行结果

```
file:///E:/机械工业/c#/daima/3/ChaixiangChu
装箱处理：值为200，装箱对象为200
```

图 3-9　实例执行结果

从图 3-9 所示的执行结果可以看出，程序只执行了装箱结果，而没有执行拆箱结果，并且在 Visual Studio 2010 中显示异常信息，具体如图 3-10 所示。



图 3-10　Visual Studio 异常提示界面

3.5　复杂的变量类型

在前面的内容中，介绍了 C# 常用的变量类型，但在现实应用中，还有其他几种常见的、比较复杂的变量类型。本节将对 C#中其他几种复杂的变量类型进行简要介绍。

3.5.1　枚举

本书前面讲解过的各种变量类型，除 string 类型外基本上都有明确的取值范围。但是在现实应用中，可能只需要变量取值范围内的一个或几个值，此时就可以使用枚举来实现。

在 C#中，使用关键字 enum 来定义枚举，具体语法格式如下：

```
enum 枚举名称:类型
{
枚举值 1,
枚举值 2,
……
枚举值 n
}
```

在应用程序内可以声明新类型的变量并赋值，具体语法格式如下：

```
类型 名;
名=枚举名称.枚举值;
```

枚举通常使用一个基本类型来存储，其默认类型是 int。枚举的基本类型有 sbyte、byte、short、ushort、uint、int、ulong 和 long。

在默认情况下，每个枚举值都会根据定义的顺序自动赋给对应的基本类型值。例如，下面的代码是按照默认顺序从 1 开始逐一递增的。

```
enum orientation : byte
  {
    Mm1 = 10,
    Mm2 = 20,
    Mm3 = 30,
    Mm4 = 40
  }
```

另外，在枚举中可以使用一个值作为另一个枚举值的基础值，并为多个枚举值设置相同的取值。看下面的代码：

```
enum orientation : byte
  {
    Mm1 = 10,
    Mm2 = Mm1,
    Mm3,
    Mm4 = Mm1
  }
```

如果某个枚举值没有赋给具体的对应值，则会自动获取一个初始值，并且此值是上个枚举值加 1。例如上面 Mm3 的值是 Mm1+1，再看下面的代码：

```
namespace aa
{
    enum aa : byte
    {
        aa1 = 2,
```

```
            aa2 = 3,
            aa3 = 4,
            aa4 = 5
        }
        class Program
        {
            static void Main(string[] args)
            {
                byte mm;
                string nn;
                aa mn;
                mn = aa.aa3;
                Console.WriteLine("myDirection = {0}", mn);
                mm = (byte)mn;
                nn = Convert.ToString(mn);
                Console.WriteLine("byte 是 = {0}", mm);
                Console.WriteLine("string 是 = {0}", nn);
                Console.ReadKey();
            }
        }
    }
```

上述实例代码的设计流程如下。

1）定义了一个枚举 aa，且设置其类型为 byte。

2）分别设置四个枚举值，并分别赋值 2、3、4、5。

3）分别定义三个变量，其中变量 mn 是枚举类型，并设置 mn 的值为 aa3 的值。

4）对枚举值 mn 进行显式转换，并通过 Convert.ToString(mn)获得枚举字符串的值。

将上述代码保存在“光盘:\ daima\3\Meiju”，执行后的显示结果如图 3-11 所示。

```
myDirection = aa3
byte 是 = 4
string 是 = aa3
```

图 3-11　实例执行结果

3.5.2　结构

结构是指由几个数组构成的数据结构，这些构成数据可以是不同的数据类型，并且根据结构可以定义需要的变量类型。在 C#中使用关键字 struct 来定义结构，具体语法格式如下：

```
struct 名 {
结构变量 1;
枚举变量 2;
……
枚举变量 n;
}
```

定义结构变量的方法和定义普通变量的方法相同。例如，在下面的代码中定义了一个结构，并在结构里定义了两个结构变量。

```
struct jiegou
  {
    orientation mm;
    double nn;
  }
```

在上述代码中，变量 mm 和 nn 是结构变量。如果需要在整个项目中能够调用上述结构变量，则可以在变量前添加关键字 public。具体代码如下：

```
struct jiegou {
    public orientation mm;
    public double nn;
  }
```

经过上述定义后，即可在整个项目中调用结构变量 mm 和 nn。

3.5.3　数组

数组是一个变量的下拉列表，通过数组可以同时存储多个数值，并且能够同时存储类型相同的值。每个数组都有自己的类型，并且数组内的各个数值都是这个类型。在 C#中声明数组的语法结构如下：

```
类型 [] 数组名;
```

其中，“类型”可以是任意变量的类型，包括本节介绍的结构和枚举类型。

数组在使用前必须初始化，例如下面的代码是错误的：

```
int [] mm;
int=5;
```

数组的初始化方式有如下两种。

1）字面值指定，字面值形式可以指定整个数组的完整内容，并且实现方法比较简单，只需使用逗号对各数组值进行分割即可。例如下面的代码：

```
int [] mm={1,3,35,6,9,100};
```

2）指定大小，即使用特定的各式指定数组的大小范围，具体格式如下：

```
类型 [] 数组名=new  类型(大小值);
```

其中，两个类型值是相同的，大小值是整数格式。例如在下面的代码中，指定了数组内有五个数值：

```
int [] mm=new int(5);
```

其中，可以组合使用字面值指定方式和指定大小方式。例如下面的代码：

```
int [] mm=new int(5)  {1,3,35,6,9};
```



在两种方式组合混用时，必须确保小括号“( )”内数组的大小和大括号“{}”内的数据个数相同。例如下面的代码是错误的:

```
int [] mm=new int(3)  {1,3,35,6,9};
```

实例 5：输出显示数组内的数据

下面通过一个具体实例来说明 C#数组的使用效果。本实例保存在“光盘:\daima\3”文件夹内，项目名为 ShiyongShuzu。实例文件的主要代码如下：

```
namespace Shuzu {
    class Program
    {
        static void Main(string[] args)
        {
            string[] Club = {"AA", "BB",
                                 "CC", "DD", "EE", "FF",
                                 "GG", "HH"};
            int i;
            Console.WriteLine("超级男声{0}强 :",
                              Club.Length);
            for (i = 0; i < Club.Length; i++)
            {
                Console.WriteLine(Club[i]);
            }
            Console.ReadKey();
        }
    }
}
```

上述实例代码的设计流程如下。

1）定义一个 string 类型的数组 Club。

2）在数组内设置八个值以分别保存八个数据。

3）通过 Club.Length 获取数组内数据的个数“8”。

实例执行后，将使用 for 循环语句将数组内的数据逐一输出，具体如图 3-12 所示。


多学一招

在使用 for 语句时必须注意它的控制范围，一旦判断错误将会造成运行错误。例如将上述代码进行如下修改:

```
for (i = 0; i <=Club.Length; i++)
```

修改后的代码运行后，将在 Visual Studio 2010 中显示异常信息，具体如图 3-13 所示。



图 3-12　实例执行结果



图 3-13　Visual Studio 2010 异常提示界面

3.6　基本.NET 框架类

.NET 框架类是整个 Visual Studio 2010 框架的核心，也是 C#的基础。在本书前面的实例应用中，已经多次用到了 Console 字符。本节将对 System 空间内常用的.NET 框架类进行简要介绍。

3.6.1　Console 类

Console 类能够为控制台应用程序提供字符的读写支持。Console 类的所有方法都是静态的，只能通过类名 Console 来调用。Console 类的常用方法有四个，分别是 WriteLine()、Write()、Read()和 ReadLine()，在下面的内容中将一一介绍。

1．方法 WriteLine()

方法 WriteLine()能够输出控制台内的指定数据，并在字符的后面自动输出一个换行符。在本书前面的实例中，已经多次用到了方法 WriteLine()。

方法 WriteLine 的具体使用格式主要有如下三种：

```
Public static void WriteLine(mm);
Public static void WriteLine(string,object);
Public static void WriteLine(string,object,object);
```

实例 6：输出显示指定的文字

下面通过一个具体的实例来说明方法 WriteLine()的使用流程。本实例保存在“光盘:\daima\3”文件夹内，项目名为 WriteLine1。实例文件的主要代码如下：

```
namespace WriteLine1{
    class Program
    {
        static void Main(string[] args)
        {
            bool zz = true;
            int i = 20;
            double mm = 10.05;
            string nn = "机械工业出版社";
            Console.WriteLine(i);
            Console.WriteLine(nn);
```

```
            Console.WriteLine("zz={0},i = {1}, mm={2}", zz, i, mm);
            Console.WriteLine("zz={0,5},i = {1,8}, mm={2,9},nn={3,-10}",
zz, i, mm, nn);
            Console.ReadKey();
        }
    }
}
```

上述实例代码的设计流程如下。

1）分别定义四个不同类型的变量，并分别赋对应值。

2）通过方法 WriteLine()将变量 i 和 nn、zz 的值输出。

3）使用格式化字符串的“{i}”格式，通过 WriteLine()方法将各变量值输出。

实例执行后，将按照顺序逐一将数据输出，具体如图 3-14 所示。

```
20
机械工业出版社
zz=True,i = 20, mm=10.05
zz= True,i =        20, mm=     10.05,nn=机械工业出版社
```

图 3-14　实例执行结果

如果输出的数据是数值类型，则还可以采用“{i,NN}”和“{i,m:NN}”格式化字符串。其中，参数“i”设置数据索引；参数“m”指定输出字符的宽度；冒号后的参数“NN”设置格式控制字符。格式控制字符的功能是设置数据值的输出格式。C#格式控制字符的具体说明如表 3-7 所示。

表 3-7　C#格式控制字符

字　符	功 能 描 述
C	设置为货币格式
D	设置为十进制格式，如果精度数大于输出数值的位数，则用 0 来补足
E	设置为科学计数法格式，精度数值指定小数的位数
F	设置为浮点格式，精度数值指定小数的位数
G	设置为一般格式，使用 E 或 F 格式的最紧凑格式
N	设置为数字格式，用逗号表示千分位，精度数值指定小数的位数
P	设置为百分数格式，精度数值指定小数的位数
X	设置为十六进制格式，如果精度数大于输出数值的位数，则用 0 来补足

例如，在下面的代码中使用了“{i,m:NN}”格式化字符串：

```
namespace WriteLine2 {
    class Program
    {
        public static void Main()
        {
            string aa = "mmnnzz";
            Console.WriteLine(aa);
```

```
                Console.WriteLine("|{0}|", aa);
                Console.WriteLine("|{0,10}|", aa);
                Console.WriteLine("|{0,-10}|", aa);

                bool bb = true;
                Console.WriteLine(bb);
                Console.WriteLine("|{0}|", bb);
                Console.WriteLine("|{0,10}|", bb);
                Console.WriteLine("|{0,-10}|", bb);

                int i = 2000;
                Console.WriteLine(i);
                Console.WriteLine("|{0}|", i);
                Console.WriteLine("|{0,10:D6}|", i);
                Console.WriteLine("|0x{0,-10:X12}|", i);
                Console.WriteLine("|{0,10:N8}|", i);

                decimal cc = 6666666.1111111m;
                Console.WriteLine(cc);
                Console.WriteLine("|{0}|", cc);
                Console.WriteLine("|{0,10:C10}|", cc);
                Console.WriteLine("|{0,-10:E8}|", cc);
                Console.WriteLine("|{0,10:F12}|", cc);
                Console.WriteLine("|{0,-10:G6}|", cc);
                Console.WriteLine("|{0,10:N8}|", cc);
                Console.WriteLine("|{0,-10:P6}|", cc);
                Console.ReadKey();
            }

        }
    }
```

上述实例代码的设计流程如下。

1）分别定义四个不同类型的变量，并分别赋对应值。

2）使用格式化字符串的“{i,m:NN}”格式，通过方法 WriteLine()将各变量值输出。

上述代码保存在“光盘:\daima\3\WriteLine2”，执行后将按顺序逐一输出数据，如图 3-15 所示。

```
mmnnzz
|mmnnzz|
|    mmnnzz|
|mmnnzz    |
True
|True|
|      True|
|True      |
2000
|2000|
|    002000|
|0x0000000007D0|
|2,000.00000000|
6666666.1111111
|6666666.1111111|
|￥6,666,666.1111111000|
|6.66666611E+006|
|6666666.111111100000|
|6.66667E+06|
|6,666,666.11111110|
|666,666,611.111110%|
```

图 3-15　实例执行结果

2. 方法 Write()

方法 Write()能够输出控制台内的指定数据，但是不能在字符的后面自动输出一个换行符。方法 Write()和方法 WriteLine()的使用方法完全一样，唯一的区别是当 Write()在输出数据时，不会在后面自动添加一个换行符。看下面的一段代码：

```
namespace Write{
    class Program
    {
        static void Main(string[] args)
        {
            bool zz = true;
            int i = 20;
            double mm = 10.5;
            string nn = "机械工业出版社";
            Console.Write(i);
            Console.Write(nn);
            Console.Write("zz={0},i = {1}, mm={2}", zz, i, mm);
            Console.Write("zz={0},i = {1}, mm={2},nn={3}", zz, i, mm, nn);
            Console.ReadKey();
        }
    }
}
```

上述实例代码的设计流程如下。

1）分别定义四个不同类型的变量，并分别赋对应值。

2）通过方法 WriteLine()将变量 i 和 nn 的值输出。

3）使用格式化字符串的“{i}”格式，通过方法 Write()将各变量值输出。

上述代码保存在“光盘:\daima\3\Write1”，执行后将按照顺序将数据输出，具体如图 3-16 所示。

```
20机械工业出版社zz=True,i = 20, mm=10.5zz=True,i = 20, mm=10.5,nn=机械工业出版社
```

图 3-16　实例执行结果

从图 3-19 所示的执行结果可以看出，当输出多行数据时，方法 Write()不能实现换行显示效果。

3．方法 Read()

方法 Read()能够从控制台的输入流中读取下一个字符，如果没有字符，则返回–1。当读操作结束后，这个方法才会被返回。如果存在可用的数据，则会读取输入流中的数据，并自动加上一个换行符作为后缀。看下面的一段代码：

```
namespace Read1{
    class Program
    {
        public static void Main()
        {
            Console.Write("输入键值并按回车键。");
            char key = (char)Console.Read();    // 显式转换!
            Console.WriteLine("输入的是: {0}", key);
```

```
                Console.ReadKey();
            }
        }
    }
```

上述实例代码的设计流程如下。

1）通过方法 Write()输出指定文本。

2）定义变量 key，将其赋值为显式转换后用户单击的键值。

3）通过方法 WriteLine()将用户输入的键值字符输出显示。

上述代码保存在“光盘:\daima\3\Readchuli”，执行后将首先显示指定的文本，如图 3-17 所示；当输入一个键值并按〈Enter〉键后，将显示用户输入的键值，具体如图 3-17 所示。



图 3-17　实例执行结果



在上述实例代码中，必须对输入键值进行显示转换，否则程序将不能正确执行。

方法 Read()只是获取一个输入的字符，如果在上述实例中输入多个键值，最终只会读取显示第一个键值字符。

4．方法 ReadLine()

方法 ReadLine()能够从控制台的输入流中读取下一行字符，如果没有字符，则返回 null。当读操作结束后，这个方法才会被返回。因为方法 ReadLine()可以返回换行符前面的整行字符，所以和方法 Read()有本质上的区别。看下面的一段代码：

```
    namespace ReadLineChuli{
        class Program
        {
            public static void Main()
            {
                Console.Write("请输键值并按回车键。");
                string mm = Console.ReadLine();
                Console.WriteLine("输入的是：{0}", mm);
                Console.ReadKey();
            }
        }
    }
```

上述实例代码的设计流程如下。

1）通过方法 Write()输出指定文本。

2）定义变量 mm，将其赋值为显式转换后的用户单击的键值。

3）通过方法 WriteLine()将用户输入的键值字符输出显示。

上述代码保存在“光盘:\daima\3\ReadLineChuli”，执行后将首先显示指定的文本，当输入一个键值并按〈Enter〉键后，将显示用户的输入键值，具体如图 3-18 所示。

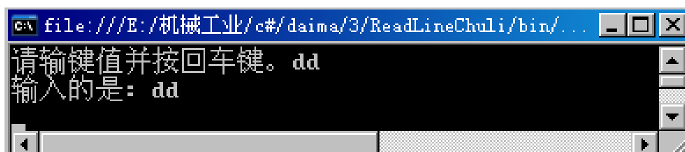


图 3-18　实例执行结果

3.6.2　Convert 类

Convert 类能够将一种基本类型转换为另一种基本类型。在 C#编程过程中，经常需要实现变量类型间的转换操作，而通过 Convert 类可以实现上述功能。

Convert 类的所有方法都是静态的，具体的使用格式如下：

```
public static 类型1 To类型(类型2 值);
```

其中，参数“类型 2”是被转换的类型；“类型 1”是要转换得到的目标类型，并且参数“To 类型”要使用 CTS 类型名称，例如类型 int 要使用 Int32，而类型 string 要使用 String。

例如，下面的代码实现了从 string 类型到 int 类型的转换。

```
public static int ToInt32(string  mm);
```

看下面的一段代码：

```
class ConvertTypes
    {
        public static void Main()
        {
            Console.Write("请输入一个数字字符串!");
            string  mm = Console.ReadLine();
            decimal nn = Convert.ToDecimal(mm);
            double  zz = Convert.ToDouble(nn);
            int i = Convert.ToInt32(zz);
            Console.WriteLine("输入的是: {0}", mm);
            Console.WriteLine("decimal形式: {0}", nn);
            Console.WriteLine("double形式: {0}", zz);
            Console.WriteLine("int形式: {0}", i);
            Console.ReadKey();
        }
    }
```

上述实例代码的设计流程如下。

1）通过方法 Write()输出指定文本。

2）分别定义 3 个变量 mm、nn 和 zz，并通过 Convert 类将各变量进行类型转换。

3）通过方法 WriteLine()将输入的字符和转换后的字符输出显示。

将上述代码保存在“光盘:\daima\3\Convert”，执行后将首先显示指定的文本，当输入一

个键值并按〈Enter〉键后，将显示此值不同转换类型后的数值，具体如图 3-19 所示。



图 3-19　实例执行结果

3.6.3　Math 类

Math 类的功能是以静态的方法提供数学函数的计算方法。例如，常见的绝对值、最大值和三角函数等。另外，Math 类还以静态成员的形式提供了 e 和π 的值。

Math 类的所有方法都是静态的，具体使用格式如下：

```
Math.函数(参数);
```

其中，参数“函数”是用于计算的数学函数；“参数”是被用来计算的数值。

看下面的一段代码：

```
class Program {
    public static void Main()
    {
        Console.Write("输入数值!");
        string aa = Console.ReadLine();
        double nn = double.Parse(aa);
        Console.WriteLine("输入的是: {0}", aa);
        Console.WriteLine("平方根 = {0}", Math.Sqrt(nn));
        Console.WriteLine("5 次幂 = {0}", Math.Pow(nn, 5.0));
        Console.WriteLine("自然对数 = {0}", Math.Log(nn));
        Console.WriteLine("正弦 = {0}", Math.Sin(nn));
        Console.WriteLine("余弦 = {0}", Math.Cos(nn));
        Console.ReadKey();
    }
}
```

上述实例代码的设计流程如下。

1）通过方法 Write()输出指定文本。

2）分别定义两个变量 aa 和 nn，并赋值输入的数字。

3）通过方法 WriteLine()将输入数字的各计算函数结果输出。

上述代码保存在“光盘:\daima\3\Math”，执行后将首先显示指定的文本，当输入一个数值并按〈Enter〉键后，将显示此值的各计算函数的结果，具体如图 3-20 所示。

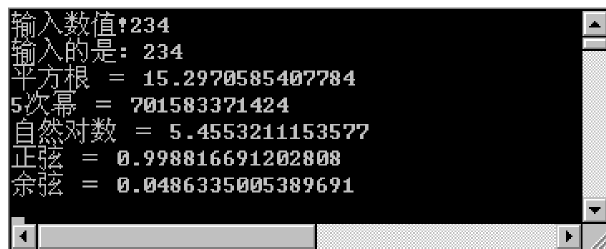


图 3-20　实例执行结果

3.7　疑难问题解析

详细介绍了 C#变量和常量的基本知识。本节将对本章中比较难以理解的问题进行讲解。

读者疑问：如何在 JavaScript 访问 C#变量?

解答：在 Java 代码里绑定 C#变量，例如:

```
<script>
var i=<%=a%>
</script>
```

其中，a 为 C#变量；i 为 JavaScript 变量。

从 DotNet 2005 以后，服务器端控件生成出来的 HTML 对象的 ID 再也不和服务器端的名称一样，而是在前面加上了前缀，所以用隐藏域的方式反而变得更麻烦了。

隐藏域取值的方法大致如下。

假设该隐藏域的 ID 为 hidden，则 JavaScript 如下:

```
var i;
var dom = document.getElementById("hidden");
if(dom!=null)i=dom.value;
```

读者疑问：请问常量和常数的区别是什么？

解答：常量是编程语言中的一个概念，而常数则是数学当中的一个概念，这是两个领域的，二者没有任何联系。例如:

```
const double pi =3.1415926
```

叫常量也可以，叫常数也可以，这只是自己的一种口语。

```
const double string txt="苦咖啡"
```

如果这个时候还叫常数，那就错了，“苦咖啡”是一个字符串，而不是一个数字。这里没有别的意义，只是教大家区分一下。



职场点拨——和客户相处之道

当开始程序员生涯后，将不可避免地要面对一些客户，并且客户的类型多种多样。也许有的读者不禁会问：程序员作为一个技术人员，只要安心做程序就好了，何须与客户交流？但是一个项目做完之后，肯定会有客户不满意的地方，有些问题是产品部的同事们所不能解决的。这就需要一线人员来解决，此时就需要根据客户的类型来一一应对了。

关于和客户的相处问题，在此总结了如下三条和客户相处的经验。

（1）提前预估客户抱怨

即使再优秀、业务和技术水平再高，也不可能消除客户的抱怨。不论工作得多么仔细，错误总难以避免。都说知己知彼百战不殆，我们需要提前预估客户的抱怨，并有针对性地做出应对计划。应该采取积极的态度对待抱怨，和那些有抱怨的客户保持好关系。让他们高兴了，将来他们会带来更多的营业额。

（2）用最快的速度解决客户的问题

如果客户提出了问题，要用最快的速度去解决。如果不能马上解决问题，也要让他们知道正在优先考虑这个问题。不要试图欺骗那些有抱怨的客户，因为那样做的结果往往比想象的要糟糕许多。以最短的时间解决客户的问题非常关键，只有让客户意识到我们已足够重视他，才能够留住这个客户。

（3）保持专业形象

当面对客户时，自始至终要表现出成熟、专业、干练的形象。有时客户会故意提出一些问题，也许他们是故意检验我们的业务能力，但客户不是专家，他们需要我们来作为他们的专家。在此期间我们一定要扮演好专家的角色，我们越专业地解决问题，就越会得到客户的信赖。

第 4 章　表达式与运算符

在本书第 3 章已经讲解了常量和变量的基本知识。从本章开始讲解 C#表达式和运算符的知识。在讲解过程中利用了前面所讲的变量和常量的知识，使读者进一步加深对这些知识的理解。通过本章能学到如下知识。

- 表达式。
- 运算符。
- 职场点拨——生活方式决定人生道路。

2006 年 X 月 X 日，天气阴

转眼间上班已经三个月了，还是做着无聊的客户服务工作，这和我当初的预期相差很远。我现在丝毫看不到自己的前途，对未来没有信心。



一问一答

小菜：我感觉前途很渺茫，丝毫看不到自己的未来。

Wisdom：呵呵，年纪轻轻就没有信心了，我在你这个年龄时可是豪情万丈啊。

小菜：哎，言归正传，本章所学的运算符和表达式有什么用？

Wisdom：本章要讲的运算符和表达式是一种对程序进行处理的处理方式。通俗一点讲，运算符和表达式就是加减乘除之类的运算符号，需要使用加法时就+，需要使用减法时就−。

4.1　表达式

使用 C#表达式，可以把变量和字面值组合起来进行特定的运算处理，以实现特定的功能。C#运算符的范围十分广泛，例如有的十分简单，而有的十分复杂，但都有一个共同点，即所有的表达式都是由运算符和被操作数构成的。具体说明如下。

1）运算符，功能是指定对特定被操作数进行什么运算，例如常用的加、减、乘、除运算。

2）被操作数，功能是指定被运算操作的对象，它可以是数字、文本、常量和变量等。

例如，下面的代码就是几个常见的表达式：

```
int i=8;
i=i*i+I;
string mm=”ab”;
string nn=”cd”;
string ff;
ff=mm+nn;
```

在 C#中，如果表达式的最终计算结果为需要类型的值，那么表达式就可以出现在需要值或对象的任意位置。例如前面例子中的表达式值，就可以通过如下代码输出：

```
System.WriteLine(mm);
System.WriteLine(nn);
System.WriteLine(ff);
System.WriteLine(Math.Sqrt(i));
```

4.2　运算符

从 4.1 节中的应用例子可以看出，处理运算符是表达式的核心。常用的运算符可以分为如下三大类。

- 一元运算符，只处理 1 个运算数。
- 二元运算符，处理 2 个运算数。
- 三元运算符，处理 3 个运算数。

在日常应用中，还可以根据被操作值的类型实现进一步划分。

4.2.1　基本运算符

在 C#中，能够实现基本操作的运算符被称为基本运算符。C#中常用的基本运算符包括如下几种。

1．点运算符“.”

通过点运算符“.”，能够对项目内不同的成员进行访问，主要包括命名空间的访问类、类的访问方法和字段等。假设在某项目中有一个名为 mm 的类，而类 mm 内有一个方法 nn，则当程序需要调用方法 nn 进行特定处理操作时，只需使用“mm.nn”语句即可实现调用。

2．括号“( )”运算符

通过括号“( )”运算符，能够定义方法和委托，并实现对方法和委托的调用。括号内可以包含需要的参数，也可以为空。例如，下面的代码使用了括号定义和调用。

```
int i=int32.Convert(“123”);
System.Console.WriteLine(“i={0}”,i);
```

3．“[ ]”运算符

“[ ]”运算符能够存储项目预访问的元素，通常用于 C#的数组处理。“[ ]”内可以为空，也可以有 1 个或多个参数。例如，下面的代码通过“[ ]”实现了数组定义和读取。

```
int [] mm=new int(3);
mm[0]=2;
mm[1]=3;
mm[2]=4;
```

4．“++”和“––”运算符

“++”和“––”运算符能够分别实现数据的递增处理和递减处理。“++”和“––”运算符支持后缀表示法和前缀表示法。例如，m++和m––的运算结果是先赋值后递增处理和先赋值后递减处理；而++m 和––m 的运算结果是先递增后赋值处理和递减后赋值处理。看下面的代码：

```
int mm=10;
int nn;
nn=mm++;
mm=10;
nn=mm––;
mm=10;
nn=++mm;
mm=10;
nn=––mm;
```

在上述代码中分别定义了 mm 和 nn 两个变量，并赋值 mm 为 10，然后对变量 nn 进行了各种类型的递增和递减处理，具体处理结果如下。

- “nn=mm++”，结果是 mm 等于 11，nn 等于 10。遵循先赋值，后加 1 的原则。
- “nn=mm++”，结果是 mm 等于 9，nn 等于 10。遵循先赋值，后减 1 的原则。
- “nn=++mm”，结果是 mm 等于 11，nn 等于 11。遵循先加 1，后赋值的原则。
- “nn=++mm”，结果是 mm 等于 9，nn 等于 9。遵循先减 1，后赋值的原则。

5．“new”运算符

“new”运算符可以创建项目中引用类型的新实例，即创建类、数组和委托的新实例。例如，下面的代码分别创建了 1 个新实例对象 mm 和新类型数组 nn。

```
object mm=new object();
int [] nn=new int[32];
```

6．“sizeof”运算符

“sizeof”运算符可以返回指定类型变量所占用的字节数。因为涉及了数量问题，所以 sizeof 只能计算值类型所占用的字节数量，并且返回结果的类型是 int。在基本类型中，“sizeof”运算符的处理结果如表 4-1 所示。

表 4-1　sizeof 运算符的处理结果

表 达 式	结　果	表 达 式	结　果	表 达 式	结　果
sizeof(byte)	1	sizeof (uint)	4	Sizeof(double)	4
sizeof (sbyte)	1	sizeof(long)	8	Sizeof(bool)	1
sizeof(short)	2	sizeof (ulong)	8	Sizeof(decimal)	16

（续）

表 达 式	结　果	表 达 式	结　果	表 达 式	结　果
sizeof (ushort)	2	Sizeof(char)	2		
sizeof(int)	4	sizeof (float)	2		



注意

“sizeof”运算符只能对类型名进行操作，而不能对具体的变量或常量进行操作。

7．“typeeof”运算符

“typeeof”运算符可以获取某类型的 System.Type 对象，它的处理对象只能是类型名或 void 关键字。如果被操作对象是一个类型名，则返回这个类型的系统类型名；如果被操作对象是 void 关键字，则返回 System.Void。



注意

“typeeof”运算符只能对类型名进行操作，而不能对具体的变量或常量进行操作。

4.2.2　数学运算符

C#中的数学运算符用于实现+、–、*、/和%等算数运算，其中包括一元运算符和二元运算符。数学运算符适用于整型、字符型、浮点型和 decimal 型。数学运算符所连接生成的表达式被称为数学表达式，其处理结果的类型是参与运算类型中精度最高的类型。

C#中数学运算符的具体信息如表 4-2 所示。

表 4-2　C#数学运算符

运 算 符	类　别	处理表达式	运 算 结 果
+	二元	mm=nn+zz	mm 的值是 nn 和 zz 的和
−	二元	mm=nn−zz	mm 的值是 nn 和 zz 的差
*	二元	mm=nn*zz	mm 的值是 nn 和 zz 的积
/	二元	mm=nn/zz	mm 的值是 nn 除以 zz
%	二元	mm=nn%zz	mm 的值是 nn 除以 zz 的余数
+	一元	mm=+nn	mm 的值等于 nn 的值
−	一元	mm=−nn	mm 的值等于 nn 乘−1 的值

实例 7：实现对指定数字的加、减、乘、除运算

下面通过一个具体的实例来说明使用 C#数学运算符的过程。本实例保存在“光盘:\daima\4”文件夹内，名为 math。主要代码如下：

```
namespace math {
    class Program {
        static void Main(string[] args) {
            double firstNumber, secondNumber;
            string userName;
```

```
                Console.WriteLine("输入标识:");
                userName = Console.ReadLine();
                Console.WriteLine("你好{0}!", userName);
                Console.WriteLine("输入一个数字:");
                firstNumber = Convert.ToDouble(Console.ReadLine());
                Console.WriteLine("输入另外一个数字:");
                secondNumber = Convert.ToDouble(Console.ReadLine());
                Console.WriteLine("{0}和{1} 的和是{2}.", firstNumber,
                          secondNumber, firstNumber + secondNumber);
                Console.WriteLine("{0}和{1}差是{2}.",
                          secondNumber, firstNumber, firstNumber –
    secondNumber);
                Console.WriteLine("{0}和{1}的积是{2}.", firstNumber,
                          secondNumber, firstNumber * secondNumber);
                Console.WriteLine("{0}除以{1}的结果是{2}.",
                          firstNumber, secondNumber, firstNumber /
    secondNumber);
                Console.WriteLine("{0}除以{1}的余数是{2}.",
                          firstNumber, secondNumber, firstNumber %
    secondNumber);
                Console.ReadKey();
            }
        }
    }
```

上述实例代码的设计流程如下。

1）分别定义变量 firstNumber、secondNumber 和 userName。

2）通过 ReadLine()分别获取用户输入的字符和数字。

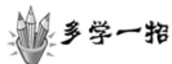
3）将用户输入的两个数字分别进行加、减、乘、除和求余计算。

4）将计算结果通过 WriteLine()输出显示。

实例执行后将首先显示指定的文本，当输入一个名字和两个数值后，将显示输入数值的各计算的结果，具体如图 4-1 所示。

```
输入标识:
AAA
你好AAA!
输入一个数字:
34
输入另外一个数字:
2
34和2 的和是36.
2和34差是32.
34和2的积是68.
34除以2的结果是17.
34除以2的余数是0.
```

图 4-1　实例执行结果



在上述实例中，通过数学运算符实现了简单的数学运算。在实际应用中，要避免运算符的溢出。算术运算符（+、–、*、/）产生的结果可能会超出所涉及的数值类型的取值范围。在一般情况下，常见溢出的具体说明如下:

- ❑ 整数算术溢出或者引发 OverflowException，或者丢弃结果的最高有效位。整数被零除总是引发 DivideByZeroException。
- ❑ 浮点算术溢出或被零除从不引发异常，因为浮点类型基于 IEEE 754，因此可以表示

无穷和 NaN（不是数字）。

- 小数算术溢出总是引发 OverflowException。小数被零除总是引发 DivideByZero Exception。

当发生整数溢出时，产生的结果取决于执行上下文，该上下文可为 checked 或 unchecked。在 checked 上下文中引发 OverflowException。在未选中的上下文中，放弃结果的最高有效位并继续执行。因此，C# 使用户有机会选择处理或忽略溢出。

除算术运算符以外，整型之间的强制转换也会导致溢出（例如，将 long 强制转换为 int）并受 checked 或 unchecked 执行的限制。然而，按位运算符和移位运算符永远不会导致溢出。

4.2.3　赋值运算符

赋值运算符可以为项目中的变量、属性、事件或索引器元素赋一个值。除了经常使用的“=”外，C#中还有其他赋值运算符。C#中常用赋值运算符的具体说明如表 4-3 所示。

表 4-3　C#赋值运算符

运 算 符	类　别	处理表达式	运 算 结 果
=	二元	mm=nn	mm 被赋予 nn 的值
+=	二元	mm+=nn	mm 被赋予 mm 和 nn 的和
−=	二元	mm−=nn	mm 被赋予 mm 和 nn 的差
*=	二元	mm*=nn	mm 被赋予 mm 和 nn 的积
/=	二元	mm/=nn	mm 被赋予 mm 除以 nn 的结果值
%=	二元	mm%=nn	mm 被赋予 mm 除以 nn 后的余数值

例如，下面两段代码的含义是相同的：

```
mm=mm+nn;
mm+=nn;
```

4.2.4　比较运算符

比较运算符可以对项目内的数据进行比较，并返回一个比较结果。在 C#中有多个比较运算符，具体说明如表 4-4 所示。

表 4-4　C#比较运算符

运　算　符	说　　明
mm= =nn	如果 mm 等于 nn，则返回 true；反之，则返回 false
mm!=nn	如果 mm 不等于 nn，则返回 true；反之，则返回 false
mm<nn	如果 mm 小于 nn，则返回 true；反之，则返回 false
mm> nn	如果 mm 大于 nn，则返回 true；反之，则返回 false
mm<= nn	如果 mm 小于或等于 nn，则返回 true；反之，则返回 false
mm >= nn	如果 mm 大于或等于 nn，则返回 true；反之，则返回 false

看下面的一段代码：

```
bool mm=5>10;
bool mm=5>=10;
bool mm=5<10;
bool mm=5<=10;
bool mm=5!=10;
```

在上述代码中，分别为变量 mm 定义了不同的值进行比较处理，具体处理结果如下。

- “mm=5>10”，结果是返回 false。
- “mm=5>=10”，结果是返回 false。
- “mm=5<10”，结果是返回 true。
- “mm=5<=10”，结果是返回 true。
- “mm=5!=10”，结果是返回 true。

4.2.5　逻辑运算符

在 4.2.4 小节的实例中，使用了类型“bool”来对数据进行比较处理。“bool”的功能是，通过返回值 true 和 false 来记录操作的结果。上述比较操作就是一种逻辑运算符，例如表 4-4 中的一些操作会返回对应的操作结果。在 C#中除了上述逻辑处理外，还有多种其他方式，具体如表 4-5 所示。

表 4-5　C#逻辑运算符

运 算 符	类　　别	处理表达式	运 算 结 果
!	一元	mm=!nn	如果 mm 等于 true，则 nn 为 false，即两者相反
&	二元	mm=nn&zz	如果 nn 和 zz 都等于 true，则 mm 为 true；否则，为 false
｜	二元	mm= nn｜zz	如果 nn 或 zz 的值等于 true，则 mm 为 true；反之，为 false
~	二元	mm= nn˜zz	如果 nn 和 zz 中只有一个值等于 true，则 mm 为 true；反之，为 false
&&	二元	mm= nn&&zz	如果 nn 和 zz 的值都等于 true，则 mm 为 true；反之，为 false
｜｜	二元	mm= nn｜｜zz	如果 nn 或 zz 的值等于 true 或都等于 true，则 mm 为 true；反之，为 false



在现实应用中，如果使用上述运算符进行复杂逻辑判断处理，应该充分注意具体情况，以避免出现因赋值失误而出现的程序运行错误。

看下面的一段代码：

```
namespace luojiyunsuan {
   class Program {
      public static void Main()
      {
         Console.Write("输入年份按 Enter 结束：");
         string str = Console.ReadLine();
         int year = Int32.Parse(str);
         bool isLeapYear = ((year % 400) == 0) || (((year % 4) == 0)
```

```
        && ((year % 100) != 0));
                Console.WriteLine("{0}是闰年：{1}", year, isLeapYear);
                Console.ReadKey();
            }
        }
    }
```

上述实例代码的设计流程如下。

1）通过 Consol()输出指定语句。

2）通过 ReadLine()获取用户输入的年份。

3）通过变量 year 进行闰年判断处理，如果是，则返回 true；反之，则返回 false。

4）将判断结果输出。

上述代码保存在“光盘:\daima\4\luojiyunsuan”，执行后将首先显示指定的文本。当输入年数“2010”后，显示对应的判断的结果，如图 4-2 所示。

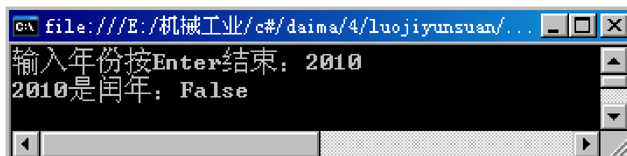


图 4-2　实例执行结果

4.2.6　移位运算符

移位运算符是指“<<”运算符和“>>”运算符，能够对指定字符进行向右或向左的移位处理，具体格式如下：

```
<<数值                            //向左移动指定数值位
>>数值                            //向右移动指定数值位
```

移位运算符的具体使用规则如下。

- 被移位操作的字符类型只能是 int、uint、long 和 ulong 中的一种，或者是通过显示转换为上述类型的字符。
- “<<”将指定字符向左移动指定位数，被空出的低位位置用 0 来代替。
- “>>”将指定字符向右移动指定位数，被空出的高位位置用 0 来代替。
- 移位运算符可以与简单的赋值运算符结合使用，组合成“<<=”和“>>＝”。具体说明如下：

1）mm<<=nn，等价于 mm=mm<<nn，即将 mm<<nn 的值转换为 mm 的类型。

2）mm>>=nn，等价于 mm=mm>>nn，即将 mm>>nn 的值转换为 mm 的类型。

4.2.7　三元运算符

三元运算符即“?:”运算符，又被称为条件运算符。其具体格式如下：

```
mm?nn:zz
```

三元运算符的运算规则如下。

1）计算条件 mm 的结果。

2）如果条件 mm 为 true，则计算 nn，计算出的结果就是运算结果。

3）如果条件 mm 为 false，则计算 zz，计算出的结果就是运算结果。

4）遵循向右扩充原则，即如果表达式为“mm?nn:zz?ff:dd”，则按照顺序“mm?nn (zz?ff:dd)”计算处理。

在使用三元运算符“?:”时，必须注意如下两点。

1）“?:”运算符的第一个操作数必须是可隐式转换为布尔类型的表达式。

2）“?:”运算符的第二个和第三个操作数决定了条件表达式的类型。具体说明如下。

- 如果 nn 和 zz 的类型相同，则这个类型是条件表达式的类型。
- 如果存在从 nn 向 zz 的隐式转换，但不存在从 zz 向 nn 的隐式转换，则 zz 类型为条件表达式的类型。
- 如果存在从 zz 向 nn 的隐式转换，但不存在从 nn 向 zz 的隐式转换，则 nn 类型为条件表达式的类型。

实例 8：判断输入的年份是否是闰年

下面通过一个具体的实例来说明使用 C#三元运算符的过程。本实例保存在“光盘:\daima\4”文件夹内，项目名为 three。其主要代码如下：

```
namespace three {
   class Program {
      public static void Main()
      {
         Console.Write("输入年份按 Enter 结束：");
         string mm = Console.ReadLine();
         int year = Int32.Parse(mm);
         bool ff = ((year % 400) == 0) || (((year % 4) == 0) &&
((year % 100) != 0));
         string yesNo = ff ? "是" : "不是";
         Console.WriteLine("{0}年{1}闰年。", year, yesNo);
         Console.ReadKey();
      }
   }
}
```

上述实例代码的设计流程如下。

1）通过 Console()输出指定语句。

2）定义变量 mm，赋值为 ReadLine()获取的用户输入的年数。

3）定义变量 year，赋值为 mm 的转换值。

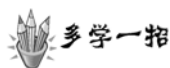
4）通过变量 ff 进行闰年判断处理。

5）定义变量 yesNo，进行三元运算处理。

6）将处理后的结果输出。

执行后将首先显示指定的文本，当输入年数“2010”后将显示对应的判断结果，如

图 4-3 所示。



输入年份按Enter结束：2010
2010年不是闰年。

图 4-3　实例执行结果

在使用三元运算符"?:"时必须注意如下两点。

1）"?:"运算符的第一个操作数必须是可隐式转换为布尔类型的表达式。

2）"?:"运算符的第二个和第三个操作数决定了条件表达式的类型。具体说明如下。

- ❑ 如果 nn 和 zz 的类型相同，则这个类型是条件表达式的类型。
- ❑ 如果存在从 nn 向 zz 的隐式转换，但不存在从 zz 向 nn 的隐式转换，则 zz 类型为条件表达式的类型。
- ❑ 如果存在从 zz 向 nn 的隐式转换，但不存在从 nn 向 zz 的隐式转换，则 nn 类型为条件表达式的类型。

4.2.8　运算符的优先级

运算符的优先级决定了表达式中运算符的执行顺序。其中，在一个表达式中的操作顺序，默认是从左到右进行计算。如果在一个表达式内有多个运算符，则必须按照它们的优先级顺序进行计算，即首先计算优先级别高的，然后计算优先级别低的。

C#中运算符的优先级顺序如表 4-6 所示。

表 4-6　C#运算符所优先级顺序

类　　别	运　算　符	左右顺序	优先级顺序
基本运算符	mm.nn、f(x)、mm++、mm––、new、typeof、checked、unchecked	从左到右	优先级顺序由高到低
一元运算符	+、–、！等	从右到左	
乘除	*、/和%	从左到右	
加减	+、–	从左到右	
移位	<<、>>	从左到右	
关系和类型检查	<、>、<=、>=、is、as	从左到右	
相等	==、!=	从左到右	
逻辑 and	&	从左到右	
逻辑 or	|	从左到右	
条件 and	&&	从左到右	
条件 or	| |	从左到右	
空合并	??	从右到左	
条件	?:	从右到左	
赋值	=、*=、/=、%=、+=、–=、<<=、>>=、&=、~=、| =	从右到左	

当在某表达式中同时出现多个同优先级运算符时，则按照左右顺序进行计算处理，并且有三个通用原则，具体如下。

1）赋值运算符外的二元与运算符都是从左向右进行计算的。

2）赋值运算符、条件运算符和空合并运算符是从右向左进行计算的。

3）有括号的要首先计算括号里面的，括号的优先级顺序是“()”>“[]”>“{}”。

4.3 疑难问题解析

本章详细介绍了 C#中运算符和表达式的基本知识。本节将对本章中比较难以理解的问题进行讲解。

读者疑问：“++”和“--”运算符能用于常量和表达式吗？

解答：“++”和“--”运算符只能用于整型变量，而不能用于常量和表达式。这很容易理解，既然常量是固定不变的，怎么能够实现递增和递减处理呢？

例如，类型 short、ushort、uint、int、ulong、long、float、double、decimal、float、char、byte、sbyte 和枚举，预定义的“++”和“--”运算符都是在此基础上进行加 1 和减 1 处理。

读者疑问：请介绍可空类型和运算符的基本知识。

解答：如果在程序中使用可空类型，就必须考虑 null 值在与各种运算符一起使用时的影响。通常可空类型与一元或二元运算符一起使用时，如果其中一个操作数或两个操作数都是 null，其结果就是 null。例如:

```
int? a = null;
int? b = a + 4;
int? c = a * 5;
```

但是在比较可空类型时，只要有一个操作数是 null，比较的结果就是 false，即不能因为一个条件是 false，就认为该条件的对立面是 true。这在使用非可空类型的程序中很常见。例如:

```
int? a = null;
int? b = -5;
if (a >= b)
   System.Console.WriteLine("a >= b");
else
   System.Console.WriteLine("a < b");
```

对于 null 值的可能性，建议避免随意比较表达式中的可空类型和非可空类型。



职场点拨——生活方式决定人生道路

虽然生活方式、生活情趣是一个人外在的行为表现，但是从深远上看，却反映了一个人对人生、对事业的态度，是道德品行和思想修养的直接体现，能够直接影响一个人的世界观，而世界观的方向又决定着一个人以后会走怎样的人生道路。究竟应该拥有什么样的生活方式呢？其答案仁者见仁智者见智，笔者在此提出如下两条建议供读者参考。

（1）要坚定信念

信念是对某人或某事信任、有信心或信赖的一种思想状态。判断一个人的生活是否正派、情趣是否健康，关键在于他的信念如何。无论是一个满脑子都是拜金主义、享乐主义和极端个人主义的人，还是一个拥有高尚的精神追求，能够全身心地投入工作和事业中的人，这都是生活情趣，如果升华一点讲，就是一种信念。我们要培养高尚的生活情趣，首先必须有坚定的理想信念，最根本的是要树立正确的世界观、人生观和价值观，加强思想改造，力争做一个对社会有用的人。

（2）要崇尚学习

中华民族是一个崇尚学习的民族，学习是一个人提高素质、做好工作的总前提。我们要把学习作为一种责任、一种精神追求、一种思想境界来认识、来对待。社会在进步，技术在发展，计算机领域的新技术更是层出不穷，我们只有多学习、多读书，不断提高自己的能力和水平，才能适应程序员这个职位。如果安于现状，则在很短时期内就将被社会所淘汰。在学习时不能只满足于一知半解、浅尝辄止，而是要坚持“学习、学习、再学习”，持之以恒地学，毫不放松地学。

作为一名程序员，我们要面对新技术、新版本的更新速度很快的问题，此时就需要我们及时学习充电，以掌握更加先进的技术。只有这样，才能够确保自己在职场中立足，获取更好的回报，人生也随之更加多彩。

第 5 章　流程控制语句

在 C#程序中，语句是应用程序完成一个指定操作的基本单位。通过语句内的变量、函数、参数和类等元素，可以实现项目中指定的功能。在默认的情况下，C#语句是从上到下顺序执行的。但有时为了实现某些特定的功能，需要设置语句按照某种特定的顺序执行。为了实现这些特定的功能，在 C#中引入了流程控制语句这一概念。通过本章能学到如下知识。

- 语句和语句块。
- 选择语句。
- 循环语句。
- 跳转语句。
- 职场点拨——创业还是就业？

2007 年 X 月 X 日，周六，天气晴

今天好累啊，刚参加了户外俱乐部的登山运动。一大早就跟着浩浩荡荡的队伍上山了。发现后山的岔路有很多，走着走着忽然发现前面的队伍停止了，我心中一阵窃喜，心想终于到顶了。可是仔细一听，原来是迷路了，面前有三道岔路，向导也不知该走哪条路……



一问一答

小菜：爬山好累啊，中途迷路了，当时感觉很害怕！

Wisdom：那你们怎么走出来的？

小菜：向导让我们在岔路口等候，他选了一条路走下去，走到一半发现错了，然后原路返回，继续走另外一条，发现又错了，原路返回，剩下那条就是正确的了！

Wisdom：还懂得用排查法来找路！你们选择路的过程还真像本章将要学的“流程控制语句”！

小菜：流程控制语句？

Wisdom：对，爬山需要选择正确的道路，而 C#程序执行时也需要选择将要执行哪条语句，这个选择过程就需要流程控制语句来实现。

5.1　语句和语句块

在 C#中，每个语句都有一个起始点和结束点，并且每个语句并不是独立的。语句是 C#

程序完成某特定操作的基本单位，能够和其他语句有着某种对应的关系。在 C#中常用的语句有如下几种。

- 空语句，只用一个分号“;”结尾。
- 声明语句，用来声明变量和常量。
- 表达式语句，由实现特定应用的处理表达式构成。
- 流程控制语句，指定应用程序内语句块的执行顺序。

例如，下面的代码都是语句：

```
int mm=5;
int nn=10;
mm=mm+nn;
Console.WriteLine(“你好，我的朋友！”)
```

在默认情况下，上述 C#语句是按照程序顺序执行的。但是，通过流程控制语句，可以指定语句执行的先后顺序。根据 C# 流程控制语句的特点，可以划分为选择语句、循环语句和跳转语句三种。

语句块是由一个或多个语句构成的，一般用大括号来分隔限定 C#语句块。在语句块内可以没有任何元素，这被称之为空块。在一个语句块内声明的局部变量或常量的作用域是块本身。C#语句块的执行规则如下。

- 如果是空语句块，则控制转到块的结束点。
- 如果不是空语句块，则控制转到语句的执行列表。

5.2　选择语句

选择语句能够从程序表达式内的多个语句中选择一个指定的语句来执行。C#中的选择语句有 if 语句和 switch 语句。本节将分别介绍上述两种选择语句的知识。

5.2.1　使用 if 语句

C#中的 if 语句即 if…else 语句，能够根据 if 后的布尔表达式的结果值来选择执行语句。if 语句的基本语法格式如下：

```
if (布尔表达式){
    处理语句;
    ……
  }
else{
    处理语句;
    ……
  }
```

其中，处理语句可以是空语句，即只有一个分号的语句；如果有处理语句或有多个处理语句，则必须使用大括号；else 子句是可选的，可以没有。

if 语句的执行流程如下。

1）首先计算 if 后的布尔表达式。

2）如果表达式的结果是 true，则执行第一个嵌套的处理语句。执行此语句完毕，将返回 if 语句的结束点。

3）如果表达式的结果是 false，并且存在 else 嵌套子句，则执行 else 部分的处理语句。执行此语句完毕，将返回到 if 语句的结束点。

4）如果表达式的结果是 false，但是不存在 else 嵌套子句，则不执行处理语句，并将返回 if 语句的结束点。

通过上述 if 语句的执行流程可以看出，不能同时执行 if 后的处理语句和 else 后的处理语句，只能执行两者之一。看下面的一段代码：

```
int aa,bb,
aa=2;
bb=3;
if (aa<bb){
  aa=3;
  bb=4;
}
else{
  aa=1;
  bb=2;
}
```

在上述代码中，首先定义了两个 int 类型的变量 aa 和 bb。然后通过 if 语句进行判断处理，具体处理如下。

- 设置布尔判断语句，通过“aa<bb”比较语句返回布尔结果。
- 如果 aa 小于 bb，则执行 if 后的处理语句，即赋值变量 aa=3，变量 bb=4。
- 如果 aa 不小于 bb，则执行 else 后的处理语句，即赋值变量 aa=1，变量 bb=2。



在编写 C# 流程控制语句时，可以在同行内同时使用多个大括号。但是为了日后的代码维护，提高代码的可观性，建议读者以缩进的方式书写大括号和代码。这样有助于避免编写代码过程中的错误。

5.2.2 使用 switch 语句

C#中的 switch 语句是多选项选择语句，能够根据测试表达式的值从多个分支选项中选择一个执行语句。switch 语句的基本语法格式如下：

```
switch (表达式) {
    case 常量表达式:
    处理语句;
    case 常量表达式:
    处理语句;
    case 常量表达式:
    处理语句;
```

```
            default:
            处理语句;
            ……
        }
```

其中，switch 后的表达式必须是 sbyte、byte、short、ushort、uint、int、ulong、long、char、string 和枚举类型中的一种，或者是可以隐式转换为上述类型的类型。case 后的表达式必须是常量表达式，即只能是一个常量值。

实例 9：根据用户的输入月份判断对应的季节

下面通过一个简单实例来说明使用 switch 语句的流程。本实例保存在“光盘:\daima\5”文件夹内，项目名为 switchxuanze。本实例的功能是根据用户输入的月份数，通过 switch 语句来判断对应的季节。实例文件的主要代码如下：

```
namespace switchxuanze
{
    class Program
    {
        public static void Main()
        {
            Console.Write("现在是几月呀？？？");
            string mm = Console.ReadLine();
            int month = Int32.Parse(mm);
            string jijie;
            switch (month)
            {
                case 12:
                case 1:
                case 2:
                    jijie = "冬";
                    break;
                case 3:
                case 4:
                case 5:
                    jijie = "春";
                    break;
                case 6:
                case 7:
                case 8:
                    jijie = "夏";
                    break;
                case 9:
                case 10:
                case 11:
                    jijie = "秋";
                    break;
                default:
```

```
                    jijie = "不存在的月份";
                    break;
                }
                Console.WriteLine("现在是{0}月，{1}天。", month, jijie);
                Console.ReadKey();
            }
        }
    }
```

上述实例代码的设计流程如下。

1）通过方法 Write()输出指定文本。

2）定义变量 mm，用于获取用户输入的月份数。

3）通过 switch 语句进行季节判断处理。

4）将比较后的结果赋值变量 jijie，然后通过 WriteLine()输出月数和对应的季节。

实例执行后将首先显示指定的文本，当输入两个数值并按〈Enter〉键后，将显示两数值的比较结果，具体如图 5-1 所示。

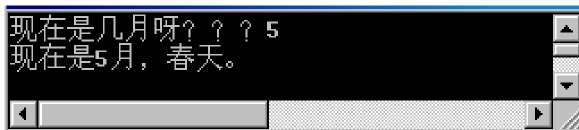
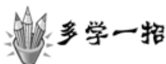


图 5-1　实例执行结果



在使用 switch 语句时应该注意如下两点。

1）必须使用 break 类的跳过语句进行处理；否则，多个 case 处理语句将会出现贯穿处理的情况，从而造成程序不能正确运行。而在 C++中，可以不用使用 break 类的跳过语句进行处理。C++可以在执行完一个 case 处理语句后，执行另外一个 case 语句。

2）当同时把多个 case 语句叠加在一起时，就会顺序执行多个 case 处理语句。

5.3　循环语句

在 C#中使用循环语句后，用户不需要编写大量的代码就可以重复执行某个指定的操作。C#中的常用循环语句有如下三种。

- while 语句。
- do…while 语句。
- for 语句。

5.3.1　使用 while 语句

在 C#中使用 while 语句后，可以通过不同的条件执行一次或多次处理语句。while 语句的基本格式如下所示：

```
while (布尔表达式)
处理语句;
```

其中，while 后的表达式必须是布尔表达式。

C#中 while 语句的执行顺序规则如下。

1）首先计算 while 后的布尔表达式。

2）如果表达式的结果是 true，则执行后面的处理语句。执行此语句完毕，将返回 while 语句的开头。

3）如果表达式的结果是 false，则返回 while 语句的结束点，循环结束。

实例 10：计算出经过多少年得到指定的目标存款

下面通过一个简单实例来说明使用 while 语句的流程，本实例保存在“5”文件夹内，项目名为 whilechuli。本实例的功能是根据用户输入的存款金额和利率，通过 while 语句计算出经过多少年得到指定的目标存款。实例文件的主要代码如下：

```
namespace whilechuli {
    class Program {
        static void Main(string[] args) {
            double cunkuan, lilu, lixicunkuan;
            Console.WriteLine("当前资金?");
            cunkuan = Convert.ToDouble(Console.ReadLine());
            Console.WriteLine("当前利率?");
            lilu = 1 + Convert.ToDouble(Console.ReadLine()) / 1000.0;
            Console.WriteLine("期望多少钱?");
            lixicunkuan = Convert.ToDouble(Console.ReadLine());
            int totalYears = 0;
            while (cunkuan < lixicunkuan)
            {
                cunkuan *= lilu;
                ++totalYears;
            }
            Console.WriteLine("存款{0} 年后将得到{2} 。",
                                 totalYears, totalYears == 1 ? "" : "s",
cunkuan);
            Console.ReadKey();
        }
    }
}
```

上述实例代码的设计流程如下。

1）通过方法 WriteLine()分别输出三段指定文本。

2）分别定义变量 cunkuan、lilu 和 lixicunkuan，用于分别获取用户输入的存款金额、利率和目标存款金额。

3）通过 while 语句执行存款处理。

4）将处理后的结果输出。

实例执行后将首先显示指定的文本，当输入三个数值并按〈Enter〉键后，将显示对应的

处理结果，具体如图 5-2 所示。

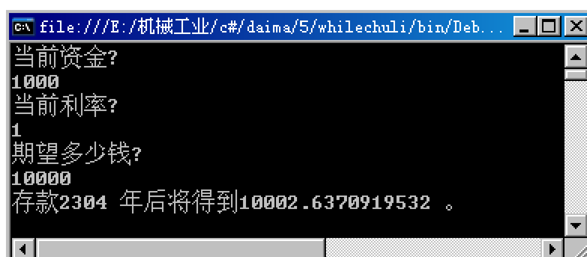


图 5-2　实例执行结果

5.3.2　使用 do…while 语句

如果 while 语句中表达式的值是 false，则不会被执行处理语句。但是有时为了满足项目的特殊要求，需要执行指定的特殊处理语句。C#中的 do…while 语句就是为了解决上述问题而定义的，它的功能是无论布尔表达式的值为多少，都要至少执行一次处理语句。

do…while 语句的基本语法格式如下：

```
do
处理语句;
while (布尔表达式)
```

do…while 语句的执行流程如下。

1）先转到 do 后面的处理语句。

2）当执行到处理语句的结束点时，计算布尔表达式。

3）如果表达式的结果是 true，则执行将返回到 do 语句的开始；否则，将来到 do 语句的结束点。

看下面的一段代码：

```
namespace zhixing{
    class Menu{
    public static void Main()
    {
        string style, order;
        do
        {
            Console.WriteLine("\n\n 选择你喜欢的专业");
            Console.WriteLine("------------------------------------\n");
            Console.WriteLine("1. 计算机");
            Console.WriteLine("2. 金融");
            Console.WriteLine("0. 退出");
            Console.Write("\n 请输入你喜欢专业的编号，按 Enter 结束!!! ");
            style = Console.ReadLine();
            if (style == "1")
```

```
            {
                do
                {
                    Console.WriteLine("\n\n 金融");
                    Console.WriteLine("------------------------------\n");
                    Console.WriteLine("1. 语文");
                    Console.WriteLine("2. 邓论");
                    Console.WriteLine("3. 马哲");
                    Console.WriteLine("0. 退出");
                   Console.Write("\n 请输入你喜欢课程的编号，按 Enter 结束!!! ");
                    order = Console.ReadLine();
                    string dish;
                    switch (order)
                    {
                        case "0":
                            break;
                        case "1":
                            dish = "语文";
                            Console.WriteLine("\n\n 你的选择是：金融—>{0}",
dish);
                            break;
                        case "2":
                            dish = "邓论";
                            Console.WriteLine("\n\n 你的选择是：金融—>{0}",
dish);
                            break;
                        case "3":
                            dish = "马哲";
                            Console.WriteLine("\n\n 你的选择是：金融—>{0}",
dish);
                            break;
                        default:
                            continue;
                    }
                } while (order != "0");
            }
            else if (style == "2")
            {
                do
                {
                    Console.WriteLine("\n\n 计算机");
                    Console.WriteLine("------------------------------
\n");
                    Console.WriteLine("1. C 语言");
                    Console.WriteLine("2. ASP");
                    Console.WriteLine("3. Java");
```

```
                        Console.WriteLine("0. 退出");
                       Console.Write("\n 请输入你喜欢学科的编号，按 Enter 结束!!! ");
                        order = Console.ReadLine();
                        string dish;
                        switch (order)
                        {
                            case "0":
                                break;
                            case "1":
                                dish = "C 语言";
                                Console.WriteLine("\n\n 你的选择是：计算机—>{0}",
        dish);
                                break;
                            case "2":
                                dish = "Java";
                                Console.WriteLine("\n\n 你的选择是：计算机—>{0}",
        dish);
                                break;
                            case "3":
                                dish = "ASP";
                                Console.WriteLine("\n\n 你的选择是：计算机—>{0}",
        dish);
                                break;
                            default:
                                continue;
                        }
                    } while (order != "0");
                }
            } while (style != "0");
            Console.WriteLine("\n\n 多谢你使用本系统，再见！ ");
        }
    }
    }
```

上述实例代码的设计流程如下。

1）通过方法 WriteLine()输出课程选择文本。

2）当用户选择喜欢的课程序号并按〈Enter〉键后，将提供对应专业下的课程。

3）当用户选择喜欢的专业下所喜欢课程的序号并按〈Enter〉键后，将显示用户对应的选择结果。

上述代码保存在“光盘\daima\5\dozhixing”，执行后将首先显示所选专业的文本提示，如图 5-3 所示；当选择某专业并按〈Enter〉键后，将显示此专业下的课程，如图 5-4 所示；当选择课程并按〈Enter〉键后，会显示对应的选择结果，具体如图 5-5 所示。

从图 5-5 所示结果可以看出，无论用户是否选择数据，都将执行 do 语句后的操作语句，即显示默认的专业选择界面。



图 5-3　选择专业



图 5-4　选择课程



图 5-5　输出最终结果

5.3.3　使用 for 语句

for 语句能够在项目中循环执行指定次数的某个语句，并维护其计数器，即计算一个初始化的表达式，并判断条件表达式的值。如果值为 true，则重复执行指定的处理语句；如果为 false，则终止循环。for 语句的基本语法格式如下：

```
for(初始化表达式；条件表达式；迭代表达式)
{
处理语句
}
```

其中，for 语句后的初始化语句可以有多个，但必须用分号“;”隔开。for 语句的执行流程如下。

1）如果有初始化的表达式，则按照初始语句的编写顺序执行它们。

2）如果有条件表达式，则计算。

3）如果没有条件表达式，则执行来到处理语句。

4）如果条件表达式的结果是 true，则执行来到处理语句。

5）如果条件表达式的结果是 false，则执行来到 if 语句的结束点。

实例 11：将指定数组内的数据从小到大顺序排列。

下面通过一个简单实例来说明使用 for 语句的具体流程，本实例保存在“光盘:\daima\5”文件夹内，项目名为 forchuli。本实例的功能是将指定数组内的数据从小到大顺序排列出来。实例文件的主要代码如下：

```
namespace forchuli{
    class Program {
        public static void Main() {
            int[] items = { 3, 5, -7, 8, 2, 1, -200, 1200, 24, 2, 7,
14 ,50 };
            for (int i = 1; i < items.Length; ++i)
            {
                for (int j = items.Length - 1; j >= i; --j)
                {
                    // 不符合排序要求则交换相邻的两个数
                    if (items[j - 1] > items[j])
                    {
                        int temp = items[j - 1];
                        items[j - 1] = items[j];
                        items[j] = temp;
                    }
                }
            }
            for (int i = 0; i < items.Length; ++i)
                Console.Write("{0} \n", items[i]);
                Console.ReadKey();
        }
    }
}
```

上述实例代码的设计流程如下。

1）定义 int 类型的数组 items，在数组内存储了十三个任意值的数字。

2）利用 for 语句进行相邻数据比较，然后将较小的数字前置。

3）将比较处理后的数据按从小到大的顺序排列。

实例执行后将数组内的数字按照从小到大的顺序排列出来，具体如图 5-6 所示。

```
-200
-7
1
2
2
3
5
7
8
14
24
50
1200
```

图 5-6　实例执行结果


多学一招

不但可以在使用 for 语句时使用 break 语句，将执行转到 for 语句的结束点位置，而且也可以使用 continue 语句将执行转到迭代表达式，然后从条件表达式开始执行 for 语句的另一次循环。例如在下面的代码中，在 for 语句内使用了 break 语句。

```
namespace zhixing
{
    class Program
    {
        public static void Main()
        {
            for (; ; )
            {
                Console.Write("请输入一个整数并按 Enter 结束!!! ");
                string mm = Console.ReadLine();
                int nn = Int32.Parse(mm);
                if (nn < 0)
                    break;
                int ji = 1;
                for (int i = 1; i <= nn; ++i)
                {
                    ji *= i;
                }
                Console.WriteLine("{0}结果等于{1}。", nn, ji);
            }
        }
    }
}
```

上述实例代码的设计流程如下。

1）通过方法 Write()输出指定的文本。

2）定义变量 mm 获取用户输入的数据。

3）定义变量 nn，并赋值 nn 转换后的数据。

4）通过 for 语句计算输入数据的“1*2*…*nn”迭代。

5）将计算结果输出。

上述代码保存在“光盘:\daima\5\break”，执行后将用户输入值的指定格式的计算结果显示出来，具体如图 5-7 所示。

上述代码的功能是根据用户输入数 mm，自动计算“1*2*…*n”的值，通过 break 语句实现了应用的跳转。

```
请输入一个整数并按Enter结束！！！2
2结果等于2。
请输入一个整数并按Enter结束！！！4
4结果等于24。
请输入一个整数并按Enter结束！！！6
6结果等于720。
请输入一个整数并按Enter结束！！！8
8结果等于40320。
请输入一个整数并按Enter结束！！！
```

图 5-7　实例执行结果

5.4　跳转语句

在 C#中，通过跳转语句可以将执行转到指定的位置，所以跳转语句常用于项目内的无条件转移控制。C#中的常用跳转语句有如下四种。

- break 语句。
- continue 语句。
- return 语句。
- goto 语句。

5.4.1 使用 break 语句

在本书前面的实例中，已经多次使用了 break 语句。C#中，只能在 switch、while、do 或 for 语句中使用 break 语句，其功能是退出当前所在的处理语句。但是，break 语句只能退出直接包含它的语句，而不能退出包含它的多个嵌套语句。

实例 12：循环显示小于 20 的整数

下面通过一个简单实例来说明使用 break 语句的具体流程。本实例保存在“光盘:\daima\5”文件夹内，项目名为 breakyuju。本实例的功能是，将指定的小于 20 的整数循环显示出来。主要代码如下：

```
namespace breakyuju {
    class Program {
        public static void Main()
        {
            for (int mm = 0; mm < 10; mm++)
            {
                Console.WriteLine("{0}次: ", mm);
             for (int nn = 0; nn < 100; nn++)
            {
                if ( nn==20 )
                    break;
                Console.Write(nn+"");

            }
            Console.WriteLine();
            Console.ReadKey();
        }
    }
    }
}
```

上述实例代码的设计流程如下。

1）定义变量 mm，利用 for 语句循环显示次数。

2）通过 WriteLine()输出次数。

3）定义变量 nn，利用 for 语句循环显示小于 20 的整数。

4）通过 WriteLine()输出结果。

上述实例代码执行后，循环显示 10 次小于 20 的整数，具体如图 5-8 所示。

由图 5-8 所示效果可以看出，break 语句只是终止了内层 for 语句，即实现了 nn 等于 20 就退出的效果，而没有对外层 for 语句造成影响。

```
0次:
012345678910111213141516171819
1次:
012345678910111213141516171819
2次:
012345678910111213141516171819
3次:
012345678910111213141516171819
4次:
012345678910111213141516171819
5次:
012345678910111213141516171819
6次:
012345678910111213141516171819
7次:
012345678910111213141516171819
8次:
012345678910111213141516171819
9次:
012345678910111213141516171819
```

图 5-8　实例执行结果


多学一招

在实际应用中，一定要注意 break 语句的终止范围——终止最近的封闭循环或它所在的 switch 语句。例如在下面代码中，条件语句包含一个应该从 1 计数到 100 的计数器，但 break 语句使得在计数达到 4 后终止循环。

```
class BreakTest {
    static void Main() {
        for (int i = 1; i <= 100; i++)
        {
            if (i == 5)
            {
                break;
            }
            Console.WriteLine(i);
        }
    }
}
```

5.4.2　使用 continue 语句

只能在 while、do 或 for 语句中使用 continue 语句。continue 语句能够忽略循环语句块内位于它后面的代码，从而直接开始另外的新的循环。但是，continue 语句只能使直接包含它的语句开始新的循环，而不能作用于包含它的多个嵌套语句。

实例 13：循环显示 0～20 的奇数

下面通过一个简单实例来说明使用 continue 语句的具体流程。本实例保存在“光盘:\daima\5”文件夹内，项目名为 continueyuju。本实例的功能是，循环显示 0～20 的奇数，主要代码如下：

```
namespace continueyuju {
    class Program {
        public static void Main()
        {
            for (int mm = 0; mm < 3; mm++)
            {
                Console.WriteLine("{0}次: ", mm);
                for (int nn = 0; nn < 20; nn++)
                {
                    if (nn%3==0)
                        continue;
                    Console.Write(nn+"\n");
                }
                Console.WriteLine();
                Console.ReadKey();
            }
        }
    }
}
```

上述实例代码的设计流程如下。

1）定义变量 nn，利用 for 语句循环显示次数。

2）通过 WriteLine()输出次数。

3）定义变量 mm，利用 for 语句得出小于 20 的奇数。

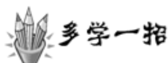
4）通过 WriteLine()输出结果。

执行上述实例代码，按下两次〈Enter〉键后，在屏幕中循环显示三次小于 20 的奇数，具体如图 5-9 所示。

a)	b)	c)
0次: 1 2 4 5 7 8 10 11 13 14 16 17 19	1次: 1 2 4 5 7 8 10 11 13 14 16 17 19	2次: 1 2 4 5 7 8 10 11 13 14 16 17 19

图 5-9 实例执行结果

由图 5-9 所示的效果可以看出，continue 语句仅仅对内层 for 语句起作用，而对外层 for 语句没有造成任何影响。



在实际应用中，经常将 continue 语句与表达式 (i < 9) 一起使用来跳过 continue 与 for 循环体末尾之间的语句。例如下面的代码:

```
class ContinueTest  {
   static void Main()
   {
      for (int i = 1; i <= 10; i++)
      {
         if (i < 9)
         {
            continue;
         }
         Console.WriteLine(i);
      }
   }
}
```

执行后将输出:

9

10

5.4.3　使用 return 语句

return 语句能够控制返回到使用 return 语句的函数成员的调用者。在 return 语句后面，可以紧跟一个可选的表达式，不带任何表达式的 return 语句只能用在没有返回值的函数成员中。所以不带表达式的 return 语句只能被用于返回类型为如下类别的对象当中。

- ❑ 返回类型是 void 的方法。
- ❑ 属性和索引器中的 set 访问器。
- ❑ 事件中的 add 和 remove 访问器。
- ❑ 实例构造函数。
- ❑ 静态构造函数。
- ❑ 析构函数。

带表达式的 return 语句只能被用在有返回值的类型中，即返回类型为如下类别的对象中：

- ❑ 返回类型不是 void 的方法。
- ❑ 属性和索引器中的 get 访问器或用户自定义的运算符。

另外，return 语句的表达式类型必须能够被隐式的转换为包含它的函数成员的返回类型。

实例 14：显示数组内的最大值

下面通过一个简单实例来说明使用 return 语句的具体流程。本实例保存在“5”文件夹内，项目名为 returnchuli。本实例的功能是显示数组内的最大值。主要代码如下所示：

```
namespace returnchuli {
   class Program {
```

```
            static int MaxValue(int[] intArray)
            {
                int mm = intArray [0];
                for (int i = 1; i < intArray.Length; i++)
                {
                    if (intArray [i] > mm)
                        mm = intArray [i];
                }
                return mm;
            }
            static void Main(string[] args)
            {
                int[] myArray = {
                1, 8, 3, 6, 2, 5, 6, 3, 0, 2,14,21,66
             };
                int mm = MaxValue(myArray);
                Console.WriteLine("最大值是: {0}!!!", mm);
                Console.ReadKey();
            }
        }
    }
```

上述实例代码的设计流程如下。

1）定义 int 类型的函数 MaxValue，用以比较数组内的数值。

2）通过 return 将得到的最大值返回给函数 MaxValue。

3）定义数组 myArray，用于存储指定的数值。

4）通过 WriteLine()将 return 返回得到的值输出。

上述实例代码执行后，将显示数组内的最大数值，具体如图 5-10 所示。

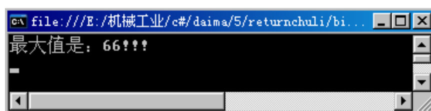


图 5-10　实例执行结果

5.4.4　使用 goto 语句

通过 goto 语句，能够将执行转到使用标签标记的处理语句中。此处的标签包括 switch 语句内的 case 标签和 default 标签，以及常用标记语句内声明的标签。例如下面的格式：

```
标签名:处理语句
```

通过上述格式声明了一个标签，这个标签的作用域是声明它的整个语句块，包括里面包含的嵌套语句块。如果里面同名标签的作用域重叠，则会出现编译错误；如果当前函数中不存在具有某名称的标签，或 goto 语句不在这个标签的范围内，也会出现编译错误。由此可见，goto 语句和前面介绍的 break 语句和 continue 语句等有很大的区别，它不但能够作用于定义它的语句块内，而且能够作用于该语句块的外部。但是，goto 语句不能将执行转到该语句所包含的嵌套语句块的内部。

5.5　疑难问题解析

本章详细介绍了 C#中流程控制语句的基本知识。本节将对本章中比较难以理解的问题进行讲解。

读者疑问：if 语句可以嵌套吗？请举例说明。

解答：在 C#应用程序内可以使用 if 语句的嵌套方式，即在一个 if 语句内使用另一个 if 语句。例如下面的书写格式:

```
  if (布尔表达式)
else{
   if (布尔表达式)
   else{
   if (布尔表达式)
   else
      处理语句;
    }
}
```

在 if 语句的嵌套语句内，布尔表达式将按照从上到下的顺序求值，直到找到条件为真时才开始执行对应的处理语句。

读者疑问：C#中有异常处理语句吗？

解答：C#的异常处理所用到的关键字是 try，它用于检查发生的异常，并帮助发送任何可能的异常。catch 用于控制权更大的方式处理错误，可以有多个 catch 子句。finally 用于在无论是否引发了异常的情况下，它的代码块都将被执行。throw 用于引发异常，可引发预定义异常和自定义异常。

C#中异常处理的格式如下:

```
try {
程序代码块;
}
catch（Exception e）{
异常处理代码块;
}
finally {
无论是否发生异常，均要执行的代码块;
}
```



职场点拨——创业还是就业?

每年的 7 月，是收获的季节，是大学生们毕业的日子。走出象牙塔的莘莘学子都会面

对毕业后是创业还是就业的选择。在选择时一定要谨慎而重视，因为这是一个决定人生命运的选择题。无论你是一名学计算机的、学网站设计的、还是学外语的、学设计类的、学法律的学生，走出大学校门后，想必都怀着对社会的美好憧憬，都想在社会上创造出自己的一片天地。

全球经济正在从金融危机的重创中缓慢复苏，当前就业压力很大，很多毕业生对自己的工作不满意，于是产生了自主创业的念头。他们走在了十字路口，不知道究竟应该走哪一条路。很多毕业生都错误地认为，只要有个好的点子，能拿到投资，再加上执着、激情和运气，就能成为下一个马云。于是在校园内到处弥漫着创业的气息，并且毕业后就马上开始了创业之路。但是现实很残酷，初出茅庐的毕业生们忽视了创业成功的真正关键更在于团队、经验和执行力，仅凭着一腔热血只能换回经验和教训。大部分创业的失败不是因为点子不好，而是因为欠缺经验，没有团队，缺乏执行力。

如果很想创业当老板，笔者对毕业生的建议是“先就业，再创业”。建议先加入一个“与自己未来想要创立的公司”同类型的公司，这样可以从中学习它是如何运营的，它的企业制度和文化是如何建立的。可以在这个公司中获得很好的培训，为自己后来的创业积累经验和财富。

但是在进入公司学习经验的时候，当拿到优厚的待遇后，很多人也习惯了朝九晚五的安稳生活，成为了一个庞大机器里的螺丝钉，失去了创业的激情和动力。随着年龄的增长，就更加喜欢随遇而安和追求稳定。所以，如果真的很确定要创业，并且也知道自己需要最合适的，最及时的、最适当的培训，也许应该找一个更好的培训场所。

第 6 章　面向对象编程技术

因为 C#是一种面向对象的编程语言，所以了解面向对象编程知识十分重要。在使用 C#编写程序时，首先应该使用面向对象的思想来分析问题，抽象出项目的共同特点。面向对象编程技术是软件开发的核心。本章将向读者简要介绍面向对象编程技术的基本知识，使读者对面向对象编程有所了解。通过本章能学到如下知识。

- 何谓面向对象。
- 统一建模语言。
- 对象建模技术。
- 面向对象分析、设计。
- 职场点拨——团队精神的重要性

2007 年 11 月 10 日，多云

上午我看了一场 NBA 比赛，球星云集的湖人队竟然输给了联盟倒数第一。看来，NBA 无弱旅啊。



一问一答

小菜：上午看球赛了吗，球星云集的湖人队竟然输给了联盟倒数第一！

Wisdom：呵呵，上午的比赛我也看了，科比太单打独斗了，很少和队友做配合。篮球是五人的运动，没有团队精神的球队输球很正常。

小菜：篮球场上团队合作很重要。那么，C#体系中最重要的是什么？

Wisdom：当然所有的知识点都很重要了，C#的应用范围太广了，精通任何一个知识点都需要花费一定的精力。

小菜：本章将要学的面向对象编程技术很重要吗？

Wisdom：作为初学者，面向对象思想很重要！一定要了解高级语言的面向对象思想，因为它的推出是软件方式的一大进步，面向对象思想已经成为当前主流语言的核心思想。

小菜：面向对象真有这么神奇吗？

Wisdom：当然了，几乎所有的高级语言都是基于面向对象诞生的，它在 C#体系中的地位犹如团体竞技项目中的团队合作。只有明白了面向对象思想，才能真正认识到 C#的强大。

6.1 何谓面向对象

本节将简要介绍面向对象编程技术的基本知识，为读者在本书后面知识的学习打下坚实的基础。

6.1.1 知道 OOP 吗

OOP（Object Oriented Programming，面向对象编程）的许多原始思想都来自于 Simula 语言，并在 Smalltalk 语言的完善和标准化过程中得到了更多的扩展和对以前的思想的重新注解。OOP 思想和 OOPL（Object Oriented Programming Language，面向对象编程语言）几乎是同步发展并相互促进的。与函数式程序设计（Functional Programming）和逻辑式程序设计（Logic Programming）所代表的接近于机器的实际计算模型相比，OOP 几乎没有引入精确的数学描叙，而是更加倾向于建立一个对象模型，它能够近似地反映应用领域内的实体之间的关系，其本质是更接近于一种人类认知事物所采用的哲学观的计算模型。

OOP 通常基于如下两种方式产生对象。

1．以原型对象为基础产生新对象

在认知心理学中，很早就使用原型概念来解释概念学习的递增特性，原型模型本身就企图通过提供一个有代表性的对象为基础来产生各种新的对象，并由此继续产生更符合实际应用的对象。而原型——委托也是 OOP 中的对象抽象，是代码共享机制中的一种。

2．以类为基础产生新对象

一个类提供了一个或多个对象的通用性描叙。从形式化的观点看，类与类型有关，所以一个类相当于从该类中产生的实例的集合。在所有同对象的世界观背景下，在类模型基础上诞生出了一种拥有元类的新对象模型，即类本身也是一种其他类的对象。

6.1.2 C#面向对象编程

面向对象编程的方法是 C#编程的指导思想。当使用 C#进行编程时，应该首先利用对象建模技术（OMT）来分析目标问题，抽象出相关对象的共性，对它们进行分类，并分析各类之间的关系；然后使用类来描述同一类对象，归纳出类之间的关系。

Coad 和 Yourdon 在对象建模技术、面向对象编程和知识库系统的基础之上，设计了一整套面向对象的方法，具体来说分为面向对象分析（OOA）和面向对象设计（OOD）。对象建模技术、面向对象分析和面向对象设计共同构成了系统设计的过程，如图 6-1 所示。

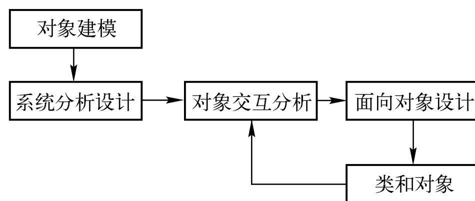


图 6-1　系统设计处理流程

6.2 统一建模语言

在对象建模、面向对象分析和设计的过程中，需要使用统一建模语言（Unified Modeling Langunianling，为 UML）来描述分析过程和结果。UML 是为了实现上述目标而设计的一种

标准通用的设计语言。本节将对统一建模语言的基本知识进行简要介绍。

6.2.1　UML 简介

UML 是一种描述性语言，它提供了多种类型的模型描述图，当在某种给定的方法学中使用这些图时，人们就能更容易理解和交流设计思想。UML 图可以分为如下三种类型。

1．静态图

静态图（Static Diagram）描述了不发生任何变化的软件元素的逻辑结构，描绘了类、对象和数据结构以及存在于它们之间的关系。

2．动态图

动态图（Dynamic Diagram）能够展示软件实体在运行期间的变化，主要描绘了执行流程、实体改变状态的方式等。

3．物理图

物理图（Physical Diagram）能够显示软件实体不变化的物理结构，主要用来描绘库文件、字节文件和数据文件，以及存在于它们之间的相互关系等。

6.2.2　类图和对象图

在 C#中，通常使用类图和对象图来表示项目内程序类的结构和各元素间的对应关系。在下面的内容中，将简要介绍类图和对象图的基本知识。

1．类图

类图表示了不同实体间的相互关系，显示了系统的静态结构。类图可以表示逻辑类，逻辑类通常指事物的种类，比如球队、电影之类的抽象描述。类图还可以表示实现类。实现类就是程序员要编写的类。实现类图与逻辑类图能够描述一些相同的类，但不会使用相同的描述属性。

通常用矩形来表示类图，并在矩形内将类分为三个部分。其中最上面的部分显示类的名称，中间部分显示类的属性，最下面的部分显示类的方法。例如图 6−2 所示的格式。

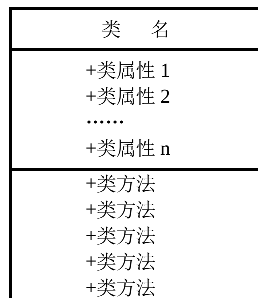


图 6−2　典型类图结构



在现实应用中，最常用、最简单的类图就是一个显示类名的长方形。在大多数的 UML 中，多数类只要有一个能够清楚表达的命名就可以了。

在类名部分可以显示类的构造类型，类的构造类型在成对双角括号“≪≫”之间指定，并放在类的名称上。常见的构造类型包括如下三类。

1）实现类：直接显示类的名称。

2）接口：在类名上方的双角括号“≪≫”内显示。

3）工具类：在类名上方的双角括号“≪≫”内显示。

如果用斜体表示类名，或者在类名下面，则表示这个类是一个抽象类。

在属性和方法的前面可以用修饰符“+”或“−”表示属性或方法的作用域，具体说明如下。

- 修饰符“-”：表示属性或方法是私有的（private）。
- 修饰符“#”：表示属性或方法是保护的（protected）。
- 修饰符“+”：表示属性或方法是公用的（public）。

通常，在类的属性或方法参数名称的冒号后，显示了属性的类型或方法的参数的类型。方法的返回值类型显示在方法后面的冒号之后。

例如，图 6-3 显示了一个类 Lei 的类图结构。

Lei

+ming: string
+xingbie: char
-nianling: int

+Chuli (in type: string): void
+Lei(in ming: string, in xingbie: char, in nianling: int): void
+Chuli2(): void
+Chuli3(): void

图 6-3　类图结构

上述类图结构对应类的 C#代码如下：

```
class Lei {
   public string ming;
   public char xingbie;
   private int nianling;
    public Lei(string ming, char xingbie, int nianling)
      {
      }
    public void Chuli3()
      {
      }
    public void Chuli2()
      {
      }
    public void Chuli (string type)
      {
      }
}
```

2．对象图

对象图能够表示类的实例化对象。通常，用一个两层的矩形结构来表示对象图，其中在上层标识对象名和类名，在下层标识对象的实例化属性值。例如在下面的代码中，创建了一个 Lei 类的对象 mm：

```
Lei mm = new Lei("mm", 'F', 24);
```

上述 mm 对象的对象图如图 6-4 所示。

mm: Lei

ming: string ="mm"
xingbie: char ='F'
nianling: int =24

图 6-4　对象图示例

在同一个应用项目中，类之间存在多种类别的相互关系，具体说明如下。

- 继承（inheritance），指一个类从其父类派生而来，继承了父类的属性和方法。基于类的继承叫做一般化（generalization），基于接口的继承叫做实现（realization）。
- 关联（association），通常用类之间的关联来表示变量实例持有对其他对象的引用，这种关系是半永久的，并没有包含关系。
- 依赖（dependency），是不同类的实例之间的暂时关系。
- 聚合（aggregation），是关联的一种特殊形式，代表着一种整体和部分的关系，并且

里面的部分可以作为其他整体的部分，而部分和整体之间也没有生命期的依赖。

- 组合（composition），是聚合的一种特殊形式。组合的关联性比聚合更强，而部分只能作为唯一的一个整体的部分，而且部分的生命周期依赖于整体的生命周期。

UML 是一种设计语言，它的目的不是表现细节而是表现结构，仅仅用于展示必要的细节。所以 UML 不可能与源代码一一对应，只存在结构上的对应关系。看下面的一段 C#代码：

```
interface zz {
}
class Lei :zz
{
   void Read(mm mm)
   {
      // …
   }
}
class Author : Lei
{
   public mm[] mms;
}
class mm
{
   public Author author;
   public Pnianling [] pnianlings;
}
class Pnianling
{
}
class mmshelf
{
   public mm[] mms;
}
```

在上面的代码中，Lei 类实现了 zz 接口，Author 类继承了 Lei 类；mm 类与 Author 类之间的关联是一对多的，即一本书可能有一个或多个作者；Lei 读书，这是一种暂时的单向依赖；mm 由一个或多个 Page（页）组成，每一个 Page（页）只能作为一本 mm 的一部分，这是一种组合关系；mmshelf 里可以不放或者放多本 mm，mm 也可以放在其他的 mmshelf 中，mmshelf 的存在并不能影响 mm 的存在性，这是一种聚合关系。

6.2.3　序列图和状态图

在 C#中，除了使用前面介绍的类图和对象图外，还可以使用序列图和状态图来表示项目内程序类的结构和各元素间的对应关系。下面将简要介绍序列图和状态图的基本知识。

1．序列图

序列图用来显示用例（或用例的一部分）的详细流程。序列图几乎是自描述的，并且显

示了流程中不同对象之间的交互关系，同时还可以很详细地显示对不同对象的各种调用。序列图有如下两个维度。

1）垂直维度以发生的时间顺序显示消息/调用的序列。

2）水平维度显示对象实例之间的交互。

对象之间的交互包括如下五种关系：

1）调用（call），一个对象调用另一个对象（或本身）的方法。

2）返回（return），返回一个值作为方法调用的结果。

3）发送（send），一个对象给另一个对象（或本身）异步发送一个消息。

4）创建（create），一个对象实例化另一个对象。

5）销毁（destroy），一个对象销毁另一个对象（或本身）。

绘制序列图的方法非常简单，图顶部的框表示类的实例对象，在框中，类实例名称和类名称之间用空格/冒号/空格来分隔。如果某个类实例向另一个类实例发送了处理消息，则绘制一条具有指向接收类实例的开箭头的连线，并把处理的名称放在连线上面。对于某些特别重要的消息，可以绘制一条具有指向发起类实例的开箭头的虚线，将返回值标注在虚线上，包括返回值的虚线，这有助于序列图的阅读。

2．状态图

状态图能够表示某个类所处的不同状态和该类的状态转换信息。虽然每个类都有状态，但每个类不一定都有一个状态图。只有当行为的改变和状态有关时才创建状态图。一般只描述在系统活动期间具有三个或更多潜在状态的类的状态图。状态图的符号集包括如下五个基本元素。

1）初始起点：使用实心圆来绘制。

2）状态转换：使用带箭头的线段来绘制。

3）当前状态：使用圆角矩形来绘制。

4）判断点：使用空心圆来绘制。

5）一个或者多个终止点：使用内部包含实心圆的圆来绘制。

在进行状态图绘制时，首先需要绘制起点和一条指向该类的初始状态的转换线段。状态本身可以在图上的任意位置绘制，然后只需使用状态转换线条将它们连接起来。

6.3　对象建模技术

对象建模技术以面向对象的思想为基础，通过对问题进行抽象而构造出一组相关的模型。通过对象建模技术，可以全面地描述问题领域的结构。对象建模技术把分析时收集到的信息构造在如下三类模型中。

- 功能模型：功能模型定义系统做什么。
- 对象模型：对象模型定义系统对谁做。
- 动态模型：动态模型定义系统如何做。

通过上述三类模型，可以从不同的角度对系统进行描述，首先分别着重于系统的某个侧面，然后整体组合起来构成对系统的完整描述。

6.3.1　功能模型

功能模型能够实现系统内部数据的传送和处理。通过功能模型，虽然可以说明从输入数据计算出什么样的输出数据，但是没有考虑参加计算的数据按什么时序来执行。功能模型是由多个数据流图组成的，它为这些数据流图指明了从外部输入、内部存储等操作，直到外部输出的整个数据流的情况。在功能模型中还包括了对象模型内部数据间的限制。

通过功能模型的数据流图可以形成一个层次结构，一个数据流图的过程可以由下一层的数据流图做进一步的说明。UML 中通常使用用例图和活动图来描述功能模型。根据面向对象思想，建立功能模型的主要步骤如下。

1）确定输入和输出值。

2）用数据流图表示功能的依赖性。

3）具体说明每个具体功能。

4）确定限制。

5）确定功能优化的准则。

上述步骤的具体实现流程如图 6−5 所示。

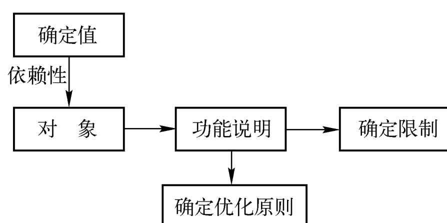


图 6−5　功能模型处理流程

6.3.2　对象模型

对象模型用于描述系统的静态结构，包括类和对象，它们的属性和操作，以及它们之间的关系。通过构造对象模型，可以发掘与项目应用所密切相关的概念。对象模型用包含对象和对象之间关系的图来表示，在 UML 中通常使用类图和对象图来描述对象模型。

使用 OMT 建立对象模型的主要步骤如下。

1）确定对象类。

2）定义数据词典，并用数据词典来描述类的属性和类之间的关系。

3）用继承来组织和简化类的结构和类之间的关系。

4）测试访问路径。

5）根据对象之间的关系和对象的功能将对象分组建立模块。

上述步骤的具体实现流程如图 6−6 所示。

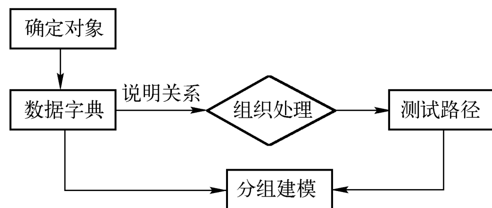


图 6−6　对象模型处理流程

6.3.3　动态模型

（1）动态模型的主要功能

动态模型的主要功能是控制系统的逻辑，检查在任何时候对象及其关系的改变，并及时描述涉及时序和改变的状态。在应用中通常使用如下两种方式来描述动态模型。

1）状态图，是状态和事件以及它们之间的关系所形成的网络，侧重于描述每一类对象的动态行为。

2）事件跟踪图，侧重于说明系统执行过程中的一个特点场景，描述完成系统某个功能

的一个事件序列。

（2）建立动态模型的步骤

建立动态模型的主要步骤如下。

1）预制典型的交互序列场景。

2）确定对象之间的事件，为每个场景建立事件跟踪图。

3）为每个系统预制一个事件流图。

4）为具有重要动态行为的类建立状态图。

5）检验不同状态图中共享的事件的一致性和完整性。

上述步骤的具体实现流程如图 6-7 所示。

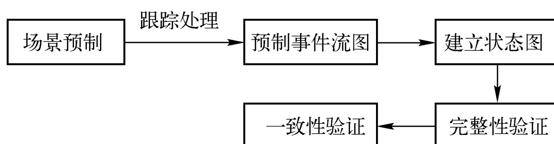


图 6-7　动态模型处理流程

6.4　面向对象分析

面向对象分析属于软件开发过程中的问题定义阶段，其目标是清晰、精确地定义问题领域。传统的系统分析产生一组面向过程的文档，定义目标系统的功能；面向对象分析则产生一种描述系统功能和问题领域的基本特征的综合文档。

6.4.1　面向对象分析的原则

C#中面向对象分析的原则和其他开发语言中的一样，需要遵循如下原则。

1．抽象

抽象是指从许多事物中舍弃个别的、非本质的特征，抽取共同的、本质性的特征的过程。

（1）抽象原则的意义

抽象是形成概念的必需手段，抽象原则具有如下两点意义。

1）虽然问题域中的事物很复杂，但分析人员并不需要了解和描述它们的一切，只需分析研究其中与系统目标有关的事物及其本质性特征即可。

2）通过舍弃个体事物在细节上的差异，抽取其共同特征而得到一批事物的抽象概念。

（2）抽象原则

抽象是面向对象方法中使用最为广泛的原则，抽象原则包括如下两个方面。

1）过程抽象：任何一个完成确定功能的操作序列，其使用者都可以把它看做一个单一的实体，尽管实际上它可能是由一系列更低级的操作完成的。

2）数据抽象：根据施加于数据之上的操作来定义数据类型，并限定数据的值只能由这些操作来修改和观察。数据抽象是面向对象分析的核心原则。它强调把数据（属性）和操作（服务）结合为一个不可分的对象，对象的外部只需要知道它做什么，而不必知道它

如何做。

2．封装

封装就是把对象的属性和服务结合为一个不可分的系统单位，并尽可能隐蔽对象的内部细节。

3．继承

特殊类的对象拥有其一般类的全部属性与服务，称做特殊类对一般类的继承。在面向对象分析中运用继承原则，就是在每个由一般类和特殊类形成的一般—特殊结构中，把一般类的对象实例和所有特殊类的对象实例都共同具有的属性和服务，一次性地在一般类中进行显式定义。在特殊类中不再重复地定义一般类中已定义的东西，但是在语义上，特殊类会自动地、隐含地拥有它的一般类（以及所有更上层的一般类）中定义的全部属性和服务。继承原则的好处是使系统模型比较简练也比较清晰。

4．分类

分类就是把具有相同属性和服务的对象划分为一类，用类作为这些对象的抽象描述。分类原则实际上是抽象原则运用于对象描述时的一种表现形式。

5．聚合

聚合能够把一个复杂的事物看成若干比较简单的事物的组装体，从而简化对复杂事物的描述。

6．关联

在思考问题时会经常用到关联这种思想方法，可以通过一个事物联想到另外的事物。能使人发生联想的原因是事物之间确实存在着某些联系。

7．消息通信

这一原则要求对象之间只能通过消息进行通信，而不允许在对象之外直接地存取对象内部的属性。通过消息进行通信是由于封装原则而引起的。在 OOA 中要求用消息连接表示出对象之间的动态联系。

8．粒度控制

当面对一个复杂的问题域时，不可能在同一时刻既能纵观全局，又能明察秋毫。此时要控制自己的视野，于是就需要遵循粒度控制原则。即当考虑全局时，要注意大的组成部分，而不能详察每一部分的具体的细节；当考虑某部分的细节时，则需要暂时撇开其余的部分。

9．行为分析

大量的事物构成现实世界中事物的行为，问题域中的各种行为，通常形成会相互依赖、相互交织的处理关系。

6.4.2　面向对象分析的阶段

面向对象分析过程可分为两个阶段，分别是问题领域分析和应用分析。

1．问题领域分析阶段

问题领域分析是软件开发的基本组成部分，能够使开发人员了解问题领域的结构，建立大致的系统实现环境。问题领域分析通常会给出一组抽象概念来作为特定系统需求开发的参考。问题领域分析实际上是一种学习过程。软件开发人员在这个阶段应该尽可能地理解当前

系统中与应用有关的知识，应该放开思维，放宽考虑的范围，尽可能地标识与应用有关的概念。问题领域分析的边界可能很模糊。有了广泛的问题领域知识，当涉及具体的应用时，就可以更快地进入状态，掌握应用的核心知识，而且在用户改变对目标系统的需求时，广泛的分析有助于预测出目标系统在哪些方面会发生哪些变化。通常进行小组分析，小组成员可以包括领域专家和分析员等。在分析过程中，应该标识出系统的基本概念（对象、类、方法、关系等）、识别问题领域的特征，并把这些概念集成到问题领域的模型中。问题领域的模型必须包含概念之间的关系，以及每个概念的全部信息。标识出来的相关概念应该根据信息内容来有机地融合到问题领域的综合视图中。

2．应用分析阶段

应用分析是依据在问题领域分析时建立起来的问题领域模型来进行的。应用分析时，把问题领域模型用于当前特定的应用之中。首先，通过收集到的用户信息来对问题领域进行取舍，把用户需求作为限制条件来使用，以缩减问题领域的信息量。因此，问题领域分析的视野大小直接影响到应用分析保留的信息量。一般来说，问题领域分析阶段产生的模型并不需要用程序设计语言来表示，而应用分析阶段产生的影响条件则需要用某种程序设计语言来表示。模型识别的要求可以针对一个应用，也可以针对多个应用。通常会着重考虑两个方面，即应用视图和类视图。在类视图中，必须对每个类的属性和操作进行细化，并表示出类之间的相互作用关系。

6.4.3　面向对象分析的具体目标

面向对象分析需要完成如下两个目标：

1）形式化地说明所面对的应用问题，最终成为软件系统基本构成的对象，以及系统所必须遵从的、由应用环境所决定的规则和约束条件。

2）明确地规定构成系统的对象如何协同工作和完成指定的功能。

通过面向对象分析所建立的系统模型被称为概念模型，因为这是以概念为中心的。概念模型由一组相关的类组成。面向对象分析可以通过自顶向下地逐层分解来建立系统模型，也可以自底向上地从已经定义的类出发，逐步构造新的类。

概念模型的构造和评审由如下五个层次构成。

- 类和对象层。
- 属性层。
- 服务层。
- 结构层。
- 主题层。

上述五个层次不是构成软件系统的层次，而是分析过程中的层次，也可以说是问题的不同侧面。每个层次的工作都为系统的规格说明增加了一个组成部分。当 5 个层次的工作全部完成时，面向对象分析的任务也就完成了。



在实际操作中，面向对象分析的目标是得出问题领域的功能模型、对象模型和动态模型，并用相应的 UML 图将它们表示出来。

6.4.4　面向对象分析的过程和步骤

通常，按照下面的步骤实现面向对象分析。

1）标识对象和类。

2）标识结构。

3）标识属性。

4）标识服务：定义服务的目的是为了定义对象的行为和对象之间的通信，具体步骤包括如下四点。

- 标识对象状态。
- 标识必要的服务。
- 标识消息连接。
- 描述服务。

5）标识主题。

6.5　面向对象设计

经过前面的分析处理后，接下来就可以进行具体设计了。面向对象设计的任务是对面向对象分析的结果作进一步的规范化整理，以便能够被面向对象编程直接接受。本节将对面向对象设计的基本知识进行简要说明。

6.5.1　面向对象设计概述

面向对象设计是一种软件设计方法，是一种工程化规范。面向对象设计的主要操作如下。

- 确定需要的类。
- 给每个类提供一组完整的操作。
- 明确地使用继承来表现共同点。

由此可见，面向对象设计的过程，就是一个根据需求决定所需的类、类的操作，以及类之间关联的过程。从面向对象分析到面向对象设计是一个逐步扩充模型的过程，两者的区别如下。

1）面向对象分析，以实际问题为中心，可以不考虑与软件实现相关的任何问题，只需考虑“做什么”的问题即可。

2）面向对象设计，是面向软件实现的实际开发活动，主要考虑“怎么做”的问题。

通过面向对象设计，可以管理程序内部各部分的相互依赖。为了实现上述功能，面向对象设计要求将程序分成块，每个块的规模应该小到可以管理的程度，然后分别将各个块隐藏在接口（interface）的后面，让它们只通过接口相互交流。比如说，如果用面向对象设计的方法来设计一个客户端—服务器应用，那么服务器和客户端之间不应该有直接的依赖，而应该让服务器的接口和客户端的接口相互依赖。

面向对象设计是一种抽象的范式，是解决软件问题的设计范式（Paradigm）。通过OOD 设计范式，可以使用对象来表现问题领域的实体，每个对象都有相应的状态和行

为。面向对象设计是一种抽象的范式，抽象可以分成很多层次，从非常概括的层次到非常特殊的层次都有，而对象可能处于任何一个抽象层次上。另外，只要对象之间有相似性，就可以把它们当成同一类的对象来处理。这些彼此不同但又互有关联的对象就可以共同构成抽象。

6.5.2　面向对象设计的原则

面向对象设计应该遵循如下原则。

1．模块化

既然是面向对象，所以很自然地应该支持模块化原则，对象就是模块。它是把数据结构和操作这些数据的方法紧密地结合在一起所构成的模块。

2．抽象

面向对象的方法不仅支持过程抽象，而且支持数据抽象。

3．信息隐藏

在面向对象方法中，信息隐藏是通过对象的封装性来实现的。

4．低联系

在面向对象方法中，因为对象是最基本的模块，所以联系主要指不同对象之间相互关联的紧密程度。低联系是设计的一个重要标准，它有助于使得系统中某一部分的变化对其他部分的影响降到最低。

6.5.3　面向对象设计的任务

面向对象设计的主要任务如下。

1．对象定义规格的求解

面向对象设计需要有一个根据设计要求进行整理和求解的过程，使之更能符合面向对象编程的需要。例如：

1）根据面向对象的概念模型整理分析所确定的对象结构、属性、方法等内容，改正错误的内容，删去不必要和重复的内容等。

2）进行分类整理，以方便下一步数据库设计和程序处理模块设计的需要。整理的方法主要是进行归类，即对类、对象、属性、方法、结构和主题进行归类。

2．数据模型和数据库设计

数据模型的设计需要确定类、对象属性的内容、消息连接的方式、系统访问、数据模型的方法等。最后，每个对象实例的数据都必须落实到面向对象的库结构模型中。

3．优化

在面向对象设计中，优化包括对象和结构的优化、抽象、集成。其优化设计过程是从另一个角度对分析结果和处理业务过程的整理和归纳。

6.5.4　面向对象设计的处理阶段

面向对象设计分为两个阶段，分别是高层设计阶段和低层设计阶段。

1．高层设计阶段

在高层设计阶段需要开发系统结构，构造软件的总体模型。具体功能描述如下。

1）标识出在具体的计算机环境中进行问题求解所需要的概念。

2）增加一批需要的类，其中包括那些能使软件系统与外部世界进行交互的类。

高层设计阶段的输出是为了适应软件要求的类以及类之间的关系，及应用子系统的视图规格说明。在高层设计的过程中，应当使子系统的高层部件之间的通信量达到最小，把子系统中相互之间存在高度交互的类划分在同一组中。

2．低层设计阶段

重复使用现有的类是面向对象技术的主要优点之一。低层设计阶段在类的详细设计阶段。设计类的目标是形成单一概念的模型（即一个概念用一个独立的类来表示），设计的部件应该是可复用的和可靠的。在类的设计过程中需要采用信息隐蔽、高内聚、低联系等设计原则。

6.5.5　面向对象设计的过程和步骤

面向对象设计的过程，是通过利用设计阶段中的五个层次来建立系统的问题领域、用户界面、任务管理和数据管理的过程。

1．问题领域

问题领域包括与所面对的应用问题直接相关的所有类和对象。实际上，在面向对象分析阶段就已经开始进行问题领域的分析。此时需要对它进行进一步的细化。面向对象分析得到了与应用有关的概念模型，面向对象设计应该对这个概念模型进行改进和增补。主要根据需求的变化来对面向对象分析阶段产生的模型中的类和对象、结构、属性和操作进行组合和分解，根据面向对象设计原则来增加必要的类、属性和关系。

2．用户界面

通常在面向对象分析阶段给出了所需的属性和操作，在面向对象设计阶段必须根据需求把交互的细节加入用户界面的设计中，包括有效的人机交互所必需的实际显示和输入。用户界面部分的设计主要由以下七个方面组成。

1）用户分类。

2）描述人及其任务的脚本。

3）设计命令层。

4）设计详细的交互。

5）继续扩展用户界面原型。

6）设计人机交互类。

7）根据图形用户界面进行设计任务管理部分的设计。

3．任务管理

任务是进程的别称，是执行一系列活动的一段程序。当系统中有许多并发行为时，需要依照各个行为的协调和通信关系来划分各种任务，以简化并发行为的设计和编码。任务管理主要包括任务的选择和调整。

4．数据管理

数据管理提供了在数据管理系统中存储和检索对象的基本结构，包括对永久性数据的访问和管理。数据管理的方法主要有三种，分别是文件管理、关系数据库管理，以及面向对象的数据库管理。

6.6 OOP 技术的应用

经过前面内容的介绍，了解了面向对象编程技术的基础知识。在接下来的内容中，将通过具体实例的实现来说明 OOP 技术的应用过程。

在 C#编程处理中，编写的 Windows 应用程序都依赖于 OOP 技术。下面将通过一个具体的 Windows 应用程序实例来说明 OOP 在其中的应用过程。本实例项目的名称为 oop，保存在“光盘:\daima\6\”文件夹内。

上述实例的具体实现过程如下。

1）打开 Visual Studio 2010，依次单击“新建”|“项目”选项，打开“新建项目”对话框。

2）在“模板”选项中选择“Windows 窗体应用程序”选项，并设置名称为 oop，如图 6-8 所示。

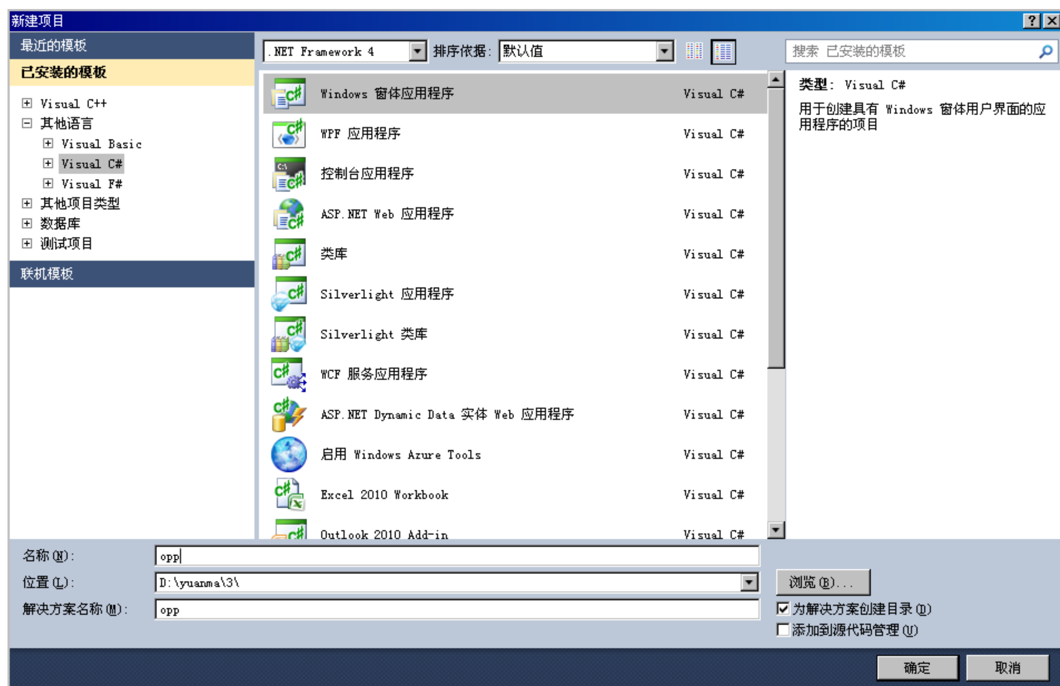


图 6-8 新建项目

3）在打开的 Visual Studio 2010 设计界面中，选择使用“工具栏”内的 Button 控件，将其放于 Form1 的中央位置，具体如图 6-9 所示。

4）双击插入的控件 button1，为其鼠标单击事件添加处理代码。修改后的代码如下：

```
namespace oop {
  public partial class Form1 : Form
  {
    public Form1()
    {
      InitializeComponent ();
```

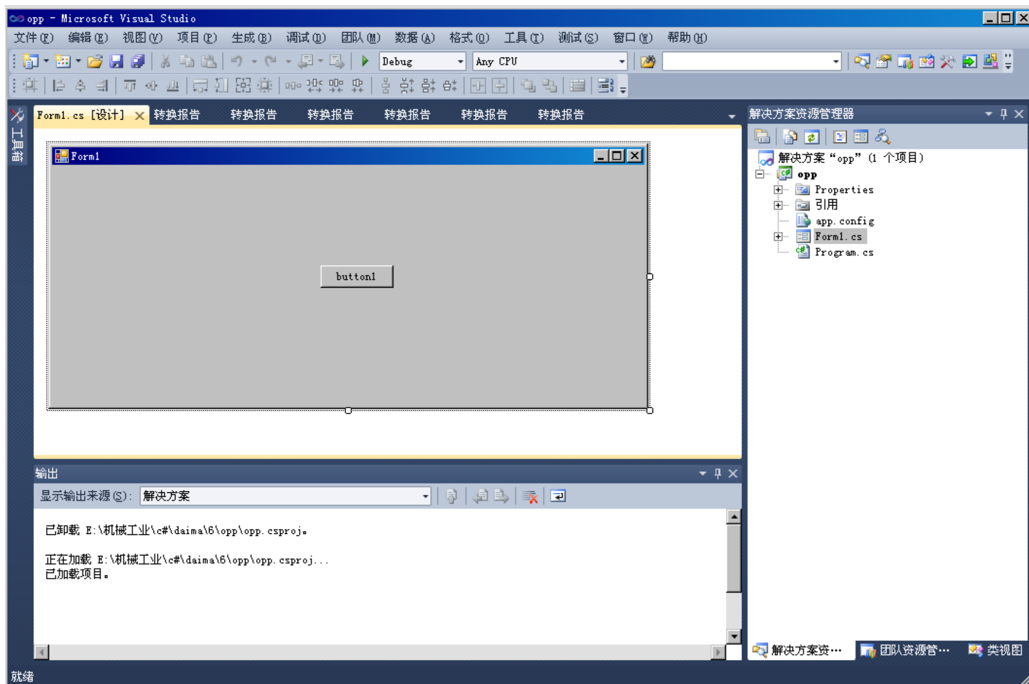


图 6-9　插入 button 控件

```
                }
                private void button1_Click(object sender, EventArgs e)
                {
                  ((Button)sender).Text = "单击了";
                  Button newButton = new Button();
                  newButton.Text = "New Button!";
                  newButton.Click += new EventHandler(newButton_Click);
                  Controls.Add(newButton);
                }
                private void newButton_Click(object sender, System.EventArgs e)
                {
                  ((Button)sender).Text = "单击了";
                }
            }
        }
```

5）整个实例设计完毕，将其保存后查看执行效果。

下面将结合面向对象编程技术，对上述实例的实现过程进行详细讲解。实例实现过程如下。

1）在 Form1 窗体内添加了一个 Button 按钮控件，这个按钮可以称之为一个对象。

2）双击添加的按钮控件，添加一个事件的处理程序 button1_Click，用于监听 Button 对象所产生的 Click 事件。具体代码如下：

```
        private void button1_Click(object sender, EventArgs e)
            {
```

```
            ((Button)sender).Text = "单击了";
            Button newButton = new Button();
            newButton.Text = "New Button!";
            newButton.Click += new EventHandler(newButton_Click);
            Controls.Add(newButton);
          }
```

上述处理程序的功能是设置当用户单击 button1 后，改变按钮上面的显示文本，并创建一个新的按钮。

3）为新创建的按钮添加一个处理程序 newButton_Click，用于监听新建 Button 对象所产生的 Click 事件。具体代码如下：

```
    private void newButton_Click(object sender, System.EventArgs e)
          {
            ((Button)sender).Text = "单击了";
          }
```

上述处理程序的功能是设置当用户单击新按钮后，将改变按钮上面的显示文本。

项目执行后，将首先在窗体内显示一个按钮，如图 6-10 所示。

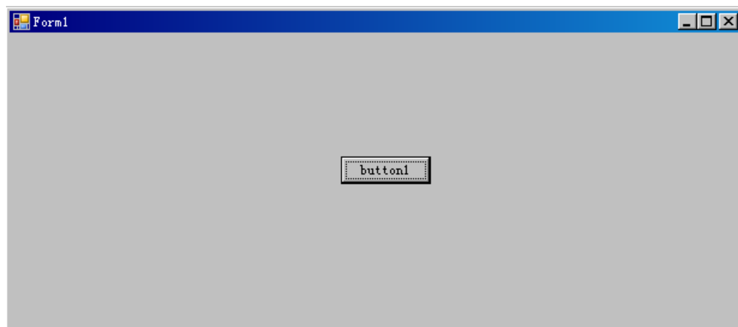


图 6-10　窗体按钮

用鼠标单击 button1 按钮后将调用处理程序，在 Form1 内创建一个新的按钮，并将 button1 按钮上的文本设置为“单击了”，具体如图 6-11 所示。

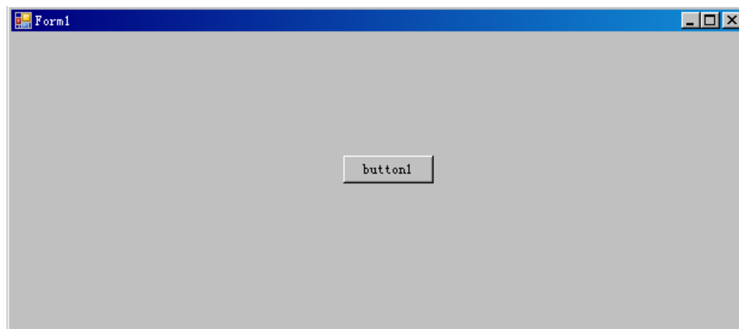


图 6-11　处理过程

单击新按钮 new 后，将会调用处理程序，将新按钮上的文本修改为“单击了”，如图 6-12

所示。



图 6-12　处理过程

从上述实例的实现过程可以看出，整个设计过程都是基于对象的，即把按钮控件作为一个对象来处理，设计独立的处理程序分别作用于这些对象。从而使整个对象独立运行，而减少了它们之间的关联，真正地实现了对象处理。

6.7　疑难问题解析

本章详细地介绍了面向对象编程技术的基本知识。本节将对本章中比较难以理解的问题进行讲解。

读者疑问：面向对象的计算模型和可计算性有何联系？

解答：编程就是用计算所需要的指令构成一种运算装置，无论程序设计思想以及程序设计语言如何发展和提高，最终人们所使用的底层计算数学模型并没有改变。但是，高级程序设计语言所带来的变革是在其语言环境中构建起了一个全新的、更抽象的虚拟计算模型。Smalltalk 引入的对象计算模型，从根本上改变了以前的传统计算模型。以前的计算模型突出的是顺序计算过程中的机器状态，而对于现在的对象计算模型所突出的对象之间的协作来说，其计算结果是由参加计算的所有的对象状态总体构成的。由于对象本身具有自身状态，也可以把一个对象看成是一个小的计算机器。这样，面向对象的计算模型就演变成了许多小的计算机器的合作计算模型。OOP 精确地抓住了计算的要点，明确了什么是可计算的，计算时间的开销和存储空间有多大。计算模型清楚地界定了可计算性的范围，从而界定了哪些问题可求解，哪些问题不可求解。虽然 OOP 为程序员提供了一种更加抽象和易于理解的新的计算模型，但是其本身并没有超越。所以不能期望 OOP 能帮助人们解决更多的问题，或者减少运算的复杂度。但 OOP 却能帮助人们用一种更容易被理解和接受的方式去描述和解决现实问题。



职场点拨——团队精神的重要性

打篮球要依靠团队的力量，不能仅靠某一个球星赢得比赛。现在从竞技赛场反思到当代

职场，在企业中的读者究竟应该如何培养和建立自己的团队素养？如何融入团队？在团队中发挥作用并获得团队及个人的成功？这些对于程序员来说都是个不小的挑战。程序员这个职业有一定的特殊性，软件行业中的个人英雄主义究竟能走多远，笔者不敢妄下结论。如何看待和对待团队的问题，是每个程序员能否走向成功的关键之一。

（1）团队精神是职场灵魂

在当代职场中，一个优秀团队既需要卓越的领导运筹帷幄，也需要优秀的成员冲锋陷阵，并且还需要高层领导、中层领导和职员拧成一股绳。都说狭路相逢勇者胜，只要团结并齐心协力，一定会创造出一个美好的未来。任何一个团结的队伍，首先要有享受团队成功的愿望和勇气，并愿意为此付出。将团队成功的喜悦当成自己的喜悦，才会发现工作是那么的有趣。懂得团队整体精神，球员才能尽情享受赛场的酣畅淋漓；有了职场精神，职员才能在工作中所向披靡。

（2）人要有精神

当代职场中硝烟弥漫，仿如一场战争，这里都是有梦想的人。要想赢得这场战争，进取与拼搏是每个人必不可少的素质。当今职场中的人们会面对日益激烈的竞争，似乎总是有层出不穷的问题来考验大家。但同时总有那么一部分人是强者，因为他们从来没有消极妥协，从来不肯轻易服输，他们积极主动、乐观向上并全力拼搏。他们也许有时会有一点犹豫，但是却决不会放弃努力。在当今职场中，只有富有激情的员工才能更加适合竞争激烈的就业现状，这样的员工也更受企业和老板的欢迎。

第二篇　核心技术篇

第7章　方　　法

C#中的方法又被称为函数，是由对象或类组成的，用于执行特定功能的代码段。方法是编程语言的核心，通过对方法的定义和调用能够实现所需要的功能。方法是函数的最基本成员，其他的函数构成成员都是以方法为基础实现的。所以从本质上讲，函数和方法几乎是等效的。本章的内容中，将详细介绍 C#中方法的基本知识，并通过对应的实例来介绍其具体使用流程。通过本章能学到如下知识。

- 定义和使用方法。
- 变量作用域。
- 静态方法与实例方法。
- 结构方法。
- 递归。
- 职场点拨——模块化设计的原则。

2007 年 X 月 XX 日，晴

武侠小说是成年人的童话。我今天看了一本精彩的武侠小说《英雄无泪》，里面的主角有一个神秘的箱子，能够根据不同的对手迅速组装出一件武器，进而迅速制敌。我也想拥有这么一个箱子。



一问一答

小菜：刚看了一本很精彩的武侠小说《英雄无泪》，我很希望拥有那个神秘的箱子。

Wisdom：呵呵，武侠毕竟是虚构的，你还是好好学习吧！

小菜：嗯，本章将要学的方法有什么用？

Wisdom：一个方法是为了实现一个功能而编写的，它正如神秘的箱子，是为了满足某个功能而设计的一段代码，这样当程序中需要实现这个功能时，在使用时直接调用此模块即可实现这个功能。

7.1　定义和使用方法

定义 C#方法的基本语法格式如下：

```
特性
修饰符 返回类型   方法名{
        方法语句
}
```

其中，特性和修饰符是可选的，修饰符用来控制方法的可访问性的范围；返回类型即方法处理结果的返回类型；方法名是定义的方法名称；方法语句是方法的核心，通过定义执行语句，设置这个方法所能够执行的功能。



方法的返回类型能够设置该方法计算和返回值的类型。方法的返回类型可以是任何的基本类型或自定义类型。如果方法中没有返回类型，则必须使用 void 作为返回类型。如果不是 void 返回类型，那么在该方法语句中必须包含一个 return 语句，并且 return 语句的返回类型必须是该类型或可以隐式地转换为该类型。

如果使用 void 作为返回类型，则在方法语句中不需要使用 return，也可以使用不带表达式的 return 语句，用来返回调用方法。

因为方法名称是一种标识符，所以它的命名应该符合标识符的命名规则。在具体使用时，最好能够在方法名称中体现出该方法的具体作用。在项目中调用已定义方法的基本语法格式如下：

```
方法名(参数)
```

使用方法非常简单，在本书前面的章节中已经使用了多次。例如在下面的代码中，定义并调用了方法 Write()：

```
namespace mm{
  class Program
    static void Write()
    {
      Console.WriteLine("这是方法!!! ");
    }
    static void Main(string[] args)
    {
      Write();
      Console.ReadKey();
    }
  }
}
```

上述代码的设计流程如下。

1）使用 static 修饰符定义一个静态方法 Write()。

2）通过 WriteLine()方法输出文本。

3）使用 Write()调用创建的方法。

上述语句执行后，将输出显示方法定义的文本。具体如图 7-1 所示。

这是方法！！！

图 7-1　实例执行结果

7.1.1　方法的返回值

通过方法的返回值可以进行最基本的数据交换，通过返回值可以计算方法的这个值。在C#中使用返回值的方式，与在表达式中使用变量计算它们包含的值相同。例如，有一个方法mm()，其返回值是一个字符串，可以使用如下代码使用这个方法：

```
string nn;
nn=mm();
```

总结前面的应用实例，可以通过如下两种方式修改方法。

1）在方法声明中指定返回值的类型，但不会使用 void 关键字。

2）使用 return 关键字结束方法的运行，并把返回值传送给调用代码。

当在项目程序内使用 return 语句时，程序会立即返回调用代码，而这个语句后的代码都不会执行。但是，return 并不一定是整个方法的最后行。比如下面的代码：

```
static double getVal()
  {
     double mm;
         if (mm<6)
       return 4.5;
       return 3.5;
  }
```

7.1.2　方法参数简介

当在方法内使用参数时，必须指定如下内容。

- 定义方法时设置的接受参数以及对应的类型。
- 在方法调用中设置的接受参数。

其中参数在方法代码中通常作为一个变量。例如在下面的代码中，通过参数作为变量进行了处理。

```
static double mm(double a, doubleb) {
        return a * b;
  }
```

实例 15：实现借钱和借给别人钱的操作

下面通过一个简单实例来说明在方法内使用参数和返回值的具体流程。本实例保存在“光盘：\daima\7”文件夹内，项目名为 hanshu。实例的功能是，通过定义的方法实现借钱和借给别人钱的处理。实例文件的主要代码如下：

```
namespace hanshu{
   class bb{
   private decimal yuanlai;
```

```
        private string name;
        public bb(string name)
        {
            this.yuanlai = 0;
            this.name = name;
            Print("我的钱数", 0);
        }
        public bool Deposit(decimal jine)
        {
            if (jine <= 0)
                return false;
            yuanlai += jine;
            Print("借了别人", jine);
            return true;
        }
        public bool Withdraw(decimal jine)
        {
            if (jine > yuanlai || jine <= 0)
                return false;
            yuanlai -= jine;
            Print("借给别人", jine);
            return true;
        }
        private void Print(string operation, decimal jine)
        {
            Console.WriteLine("名字: {0}", name);
            Console.WriteLine("{0}: {1}", operation, jine);
            Console.WriteLine("现在金额: {0}", yuanlai);
            Console.WriteLine("---------------------------");
            Console.ReadKey();
        }
        public static void Main()
        {
            bb acc = new bb("我是善良的人");
            bool succeed = acc.Deposit(10000);
            if (!succeed)
                Console.WriteLine("aaaaa! ");
            succeed = acc.Withdraw(5000);
            if (!succeed)
                Console.WriteLine("bbb! ");
        }
    }
    }
```

上述实例代码的设计流程如下。

1）定义方法 bb()，设置最初拥有的钱数为 0，并调用方法 Print()输出指定的文本字符。

2）定义借钱处理方法 Deposit()，并调用方法 Print()输出指定的文本。

3）定义借出钱处理方法 Withdraw()，并调用方法 Print()输出指定的文本。

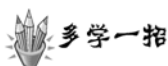
4）定义 Main()，设置用户的名字为“我是善良的人”，最初钱数是 0。

5）定义变量 succeed，判断处理钱数是否合法。

实例执行后将首先显示最初的钱数，即为 0；当按〈Enter〉键后，将执行借钱处理方法，并显示处理后的钱数；再次按〈Enter〉键后，将执行借出钱处理方法，并显示处理后的钱数。最终执行结果如图 7-2 所示。

```
名字：我是善良的人
我的钱数：0
现在金额：0
---------------------------
名字：我是善良的人
借了别人：10000
现在金额：10000
---------------------------
名字：我是善良的人
借给别人：5000
现在金额：5000
---------------------------
```

图 7-2　实例执行结果



在上述实例中，如果将借出的钱数修改为 11000，则会输出“不借给别人”的提示。另外，通过一个简单的方法实现了借钱和借给别人钱的事物处理。但是在方法内使用参数时，读者必须指定如下内容。

- ❑ 定义方法时设置的接受参数以及对应的类型。
- ❑ 在方法调用中设置的接受参数。

7.2　变量作用域

只能从代码的本地作用域内访问变量。在方法内的变量都有其对应的作用域，当访问这个变量时，需要通过这个作用域来实现。一般来说，某方法的变量只能在这个方法中使用。看下面的一段代码：

```
namespace zuoyongyu{
  class Program
  {
    static void Write()
    {
       string mm="大家好，我是方法!!! ";
       Console.WriteLine("mm={0}",mm);
    }
```

```
            static void Main(string[] args)
            {
                Write();
                Console.ReadKey();
            }
        }
    }
```

在上述代码的设计流程如下。

1）使用 static 修饰符定义一个静态方法 Write()。

2）在方法内定义 string 类型的变量 mm，并赋值给对应的文本值。

3）使用 Write()调用创建的方法，将变量值输出。

上述语句执行后，将输出显示方法内变量定义的文本，具体如图 7-3 所示。

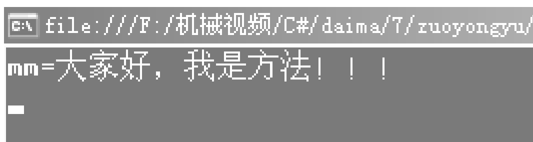


图 7-3　实例执行结果

如果将上述代码中的变量 mm 在方法 Main()内定义，程序执行后将会出现运行错误。

7.3　静态方法与实例方法

在 C#中，使用 static 修饰的方法被称为静态方法，而没有用 static 修饰的方法被称为实例方法。静态方法的成员比较特殊，不属于一个类的某个具体实例或对象，而是属于类本身。在静态方法中不能使用 this 关键字，并且只能访问类中的静态成员，而不能使用实例成员。在项目文件中访问静态方法时只能使用类名，而不需要创建额外的对象。

实例方法可以使用类的所有成员，在调用实例方法时必须使用类的实例或对象来引用，并且在实例方法内可以使用 this 关键字。

实例 16：实现进货和出货处理

下面通过一个简单实例来说明使用实例方法的具体流程。本实例代码保存在“7”文件夹内，项目名为 jingdong。本实例的功能是通过方法实现进货和出货处理。实例文件的主要代码如下：

```
    namespace jingdong{
        class Account
        {
            private decimal kucun;
            private string name;
            public Account(string name)
            {
                this.kucun = 0;
```

```
        this.name = name;
        Print("店名", 0);
    }
    public bool Deposit(decimal shuliang)
    {
        if (shuliang <= 0)
            return false;
        kucun += shuliang;
        Print("进货", shuliang);
        return true;
    }
    public bool Withdraw(decimal shuliang)
    {
        if (shuliang > kucun || shuliang <= 0)
            return false;
        kucun -= shuliang;
        Print("出货", shuliang);
        return true;
    }
    private void Print(string operation, decimal shuliang)
    {
        Console.WriteLine("店名: {0}", name);
        Console.WriteLine("{0}: {1}", operation, shuliang);
        Console.WriteLine("库存: {0}", kucun);
        Console.WriteLine("---------------------------");
        Console.ReadKey();
    }
    public static void xinxi()
    {
        Console.WriteLine(" 网上×××店, 电话: 12345678");
        Console.WriteLine("---------------------------");
        Console.WriteLine("    顾客至上    服务第一    ");
        Console.WriteLine("---------------------------");
    }
    public static void Main()
    {
        Account.xinxi();
        Account acc = new Account("我的商店");
        bool succeed = acc.Deposit(100);
        if (!succeed)
            Console.WriteLine("进货失败! ");
        succeed = acc.Withdraw(50);
        if (!succeed)
            Console.WriteLine("出货失败! ");
            Console.ReadKey();
    }
```

```
        }
    }
```

上述实例代码的设计流程如下。

1）定义方法 Account()，设置商店最初库存为 0，并调用方法 Print()输出指定的文本。

2）定义进货处理方法 Deposit()，并调用方法 Print()输出指定的文本。

3）定义出货处理方法 Withdraw()，并调用方法 Print()输出指定的文本。

4）定义 Main()，设置用户的名称为“我的商店”，最初库存是 0。

5）定义变量 succeed，判断处理货量是否合法。

执行后将首先显示最初的商品数，即为 0；当按〈Enter〉键后，将执行进货处理方法，并显示处理后的商品数；再次按〈Enter〉键后，将执行出货处理方法，并显示处理后的商品数。最终执行结果如图 7-4 所示。

在上述实例中，通过定义变量 succeed 判断处理货量是否合法。例如，将出货数修改为 100，则会输出“出货失败！”提示，如图 7-5 所示。

```
网上×××店，电话：12345678
-----------------------------
    顾客至上    服务第一
-----------------------------
店名：我的商店
店名：0
库存：0
-----------------------------
店名：我的商店
进货：100
库存：100
-----------------------------
店名：我的商店
出货：50
库存：50
-----------------------------
```

图 7-4　实例执行结果

```
网上×××店，电话：12345678
-----------------------------
    顾客至上    服务第一
-----------------------------
店名：我的商店
店名：0
库存：0
-----------------------------
店名：我的商店
进货：100
库存：100
-----------------------------
出货失败！
```

图 7-5　实例执行结果

上面的方法 xinxi()是静态方法，所以在使用时直接使用类名来调用；而方法 Deposit()和方法 Withdraw()则是实例方法，在调用时必须创建实例对象。



多学一招

对于 C#中的方法来说，是没有分配内存这个概念的。无论静态方法还是动态方法，都是存放在某个特定的位置上的。不同的是在类加载的时候，静态方法通过类名引用，动态方法通过实例化引用。而实例化类时，不会为类的方法分配内存，只是给类实例的数据分配内存，所以成员变量和成员方法的内存分配是不一样的。

静态是类的，实例是对象的，即静态成员就是类所有对象所具备的特点，而实例成员是每个对象自有的实现，是不一样的。例如，每个人都有头发、眼睛、耳朵、名字、性别、年龄、身高等，这些都是静态的，但是每个人的这些属性是不一样的，这些不同的属性就是实例化。所以也可以说，静态方法不需要依赖类中的属性，静态方法本身就可以封闭地完成一个功能，而实例的方法更多会用到类中的属性。静态方法在它一启动的时候就实例化了，因而静态内存是连续而有限制的；实例方法则在程序运行中生成内存，实例申请的是离散的空间。静态方法大多用于数据连接上。

7.4　方法参数详解

C#的方法可以有参数，也可以没有参数。由前面的知识可以了解到，当有多个参数时需要使用逗号“,”隔开。在 C#程序中，可以根据具体需要而使用不同类的参数。具体来说，C#方法的参数可以分为如下四类。

1）值参数：不使用任何修饰符声明。

2）引用参数：使用 ref 修饰声明。

3）输出参数：使用 out 修饰声明。

4）参数组：使用 params 修饰声明。

7.4.1　使用值参数

值参数是指当声明时不使用任何修饰字符的参数。在 C#中，一个值参数相当于一个局部变量，并且在程序声明和调用使用时，值参数只能将值带进方法，而不能将值带出方法。看下面一段代码：

```
namespace zhican
{
    class Program
    {
        static void mm(int x, int y)
        {
            Console.WriteLine("进入 mm 方法时：x = {0}, y = {1}", x, y);
            int temp = x;
            x = y;
            y = temp;
            Console.WriteLine("退出 mm 方法时：x = {0}, y = {1}", x, y);
        }
        static void Main()
        {
            int i = 1, j = 2;
            Console.WriteLine("执行 mm 方法前：i = {0}, j = {1}", i, j);
            mm(i, j);
            Console.WriteLine("执行 mm 方法后：i = {0}, j = {1}", i, j);
            Console.ReadKey();
        }
    }
}
```

上述实例代码的设计流程如下。

1）定义方法 mm()，并为它设置两个 int 类型的参数 x 和 y。

2）将 x 和 y 的值进行赋值处理，并交换它们的值。

3）定义方法 Main()，并分别定义 int 类型的变量 i 和 j。

4）将变量 i 和 j 作为方法 mm()的参数进行处理。

5）通过 WriteLine()输出对应的处理结果。

上述代码保存在“光盘：\daima\7”文件夹内，项目名为 zhican。执行后，将调用方法，进行处理并输出对应的结果，具体如图 7-6 所示。

```
执行mm方法前：i = 1, j = 2
进入mm方法时：x = 1, y = 2
退出mm方法时：x = 2, y = 1
执行mm方法后：i = 1, j = 2
```

图 7-6　实例执行结果

由图 7-6 的执行结果可以看出，在方法 mm()内成功地交换了参数 x 和 y 的值，但是不会影响在方法 Main()内调用方法 mm()时所使用的实际参数 i 和 j 的值。

7.4.2　使用引用参数

在 C#中，使用 ref 修饰符声明的参数是引用参数，一个引用参数不能创建新的存储位置。引用参数表示的存储位置在方法调用中，被作为实际参数给出变量所表示的存储位置。在声明和调用使用程序时，通过引用参数可以将值带进方法内，也可以将值带出方法而在方法外使用。

实例 17：交换方法内的参数值

下面通过一个简单实例来说明在方法内引用参数的具体流程。本实例保存在“光盘：\daima\7”文件夹内，项目名为 yincan。实例的功能是将方法内的参数值做交换处理。实例文件的主要代码如下：

```
namespace yincan
{
    class Program
    {
        static void mm(ref int x, ref int y)
        {
            Console.WriteLine("进入方法 mm 时：x = {0}, y = {1}", x, y);
            int temp = x;
            x = y;
            y = temp;
            Console.WriteLine("退出方法 mm 时：x = {0}, y = {1}", x, y);
        }
        static void Main()
        {
            int i = 1, j = 2;
            Console.WriteLine("执行方法 mm 前：i = {0}, j = {1}", i, j);
            mm(ref i, ref j);
            Console.WriteLine("执行方法 mm 后：i = {0}, j = {1}", i, j);
            Console.ReadKey();
        }
```

```
        }
    }
```

上述实例代码的设计流程如下。

1）定义方法 mm()，并为它设置两个 int 类型的引用参数 x 和 y。

2）将 x 和 y 的值进行赋值处理，并交换它们的值。

3）定义方法 Main()，并分别定义 int 类型的变量 i 和 j。

4）将变量 i 和 j 作为方法 mm()的引用参数进行处理。

5）通过 WriteLine()输出对应的处理结果。

上述实例代码执行后，将调用方法，进行处理并输出对应的结果，具体如图 7-7 所示。

```
执行方法mm前：i = 1, j = 2
进入方法mm时：x = 1, y = 2
退出方法mm时：x = 2, y = 1
执行方法mm后：i = 2, j = 1
```

图 7-7　实例执行结果

由图 7-7 的执行结果可以看出，在方法 mm()内成功地交换了参数 x 和 y 的值。当在方法 Main()内调用方法 mm()时，也实现了实际参数 i 和 j 的值的交换。

7.4.3　使用输出参数

在 C#中，使用 out 修饰符声明的参数是输出参数，一个输出参数不能创建新的存储位置。输出参数表示的存储位置在方法调用中被作为参数给出变量所表示的存储位置。在声明和调用使用程序时，输出参数不能将值带进方法内，而只能将值带出方法外使用。

在方法内部，输出参数最初是被赋值的，即对应参数已经被明确赋值，这和局部变量是一样的。所以，在使用输出参数时，必须在使用前进行明确的赋值。看下面的一段代码：

```
static void mm(out int x, out int y)
{
    int temp = x;
    x = y;
    y = temp;
}
static void Main()
{
    int i = 1, j = 2;
    mm(out i,out j);
}
```

在上述代码中，方法 mm()对变量 temp、x 和 y 的值进行赋值处理会出现错误。因为参数 x 和 y 是输出参数，在使用时必须先赋值。而在方法 Main()内，变量 i 和 j 被赋了初始值，当把两边作为方法 mm()的输出参数时，程序不会出现错误，并且赋的 i 和 j 值将会被忽略。

7.4.4 使用参数数组

在 C#中，使用 params 修饰符声明的参数是参数数组，不能把修饰符 params、ref 或 out 组合使用。在调用方法时，可以通过如下两种方式为参数数组指定对应的参数。

1）第一种：赋给参数数组的实参是一个表达式，其类型可以隐式转换为参数数组的类型。这样，参数数组将和值参数完全一致。

2）第二种：可以为参数数组设置指定个数的实参，可以是 0 个。其中的每个实参是一个表达式，其类型可以隐式转换为参数数组元素的类型。在上述情况下，调用时会创建一个参数数组类型的实例，其包含的元素个数等于给定的参数的数量，再用给定的参数数值初始化此数组实例的每个元素，然后将创建的新数组实例用做实参。

在调用方法时，可以使用可变数量的实参，参数数组与同类型的值参数完全等效。如果在方法内改变形参的值，对应的实参值不会造成影响。

如果形参表包含一个参数数组，则该参数数组必须位于该列表的最后，而且必须是一维数组类型。例如 string[]类型和 string[][]类型可以作为参数数组，而 string[,]类型则不能。

实例 18：输出指定数组内的存储数据

下面通过一个简单实例来说明使用参数数组的具体流程，本实例保存在“光盘：\daima\7”文件夹内，项目名为 canshuzu。实例的功能是通过方法输出指定数组内的存储数据。实例文件的主要代码如下：

```
namespace canshuzu
{
    class Program
    {
        static void mm(params int[] args)
        {
            Console.Write("调用方法mm：数组内包含{0}个元素，分别是:", args.Length);
            for (int i = 0; i < args.Length; ++i)
                Console.Write(" {0} |", i);
            Console.WriteLine();
        }
        static void Main()
        {
            int[] arr = { 1, 2, 3 };
            mm(arr);
            mm(10, 20, 30, 40);
            mm();
            Console.ReadKey();
        }
    }
}
```

上述实例代码的设计流程如下。

1）定义方法 mm()，通过 Write()输出指定文本。

2）通过 for 语句逐一读取数组内的存储数据。

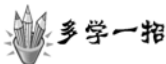
3）定义数组 arr，并为存储的数据赋值。

4）调用方法 mm()，使数组作为方法的参数。

上述实例代码执行后，将调用方法并输出对应的处理结果，具体如图 7-8 所示。

```
调用方法mm：数组内包含3个元素，分别是：0 | 1 | 2 |
调用方法mm：数组内包含4个元素，分别是：0 | 1 | 2 | 3 |
调用方法mm：数组内包含0个元素，分别是：
```

图 7-8　实例执行结果



在上述实例中，通过定义的方法输出了数组内的存储数据。在具体使用时读者可以把握如下两点。

1）如果方法的参数列表中除了参数数组外还需要包含其他参数，参数数组必须为列表中的最后一项。

2）如果在方法的参数中采用数组，但不用 params 修饰，则其代表参数必须是一个数组，而不是可选数量的同类参数。

7.4.5　使用数组参数

在 C#项目中，数组可以作为值、引用或输出参数传递给方法。并且在使用过程中，可以根据具体要求灵活使用。当使用数组参数时，必须遵循如下三个原则。

1．作为值参数

当把数组作为值参数传入时，在传入数组前必须创建数组对象，并且传入前的数组元素值可以传入方法，具体如下。

1）如果方法内没有改变数组对象的值，而是仅仅改变了数组元素的值，则在方法内修改的数组元素值可以从方法内带出。

2）如果方法内改变了数组对象的值，则在方法内修改的数组元素值不会从方法内带出。

2．作为引用参数

当把数组作为引用参数传入时，在传入数组前必须创建数组对象。传入前的数组元素值可以传入方法，而在方法内修改的数组元素值也可以从方法内带出。

3．作为输出参数

当把数组作为输出参数传入时，在传入数组前可以创建数组对象，也可以不创建数组对象。如果在传入数组前创建了数组对象，则传入方法后会忽略该数组对象的值。所以只有在方法内创建数组对象时，才可以从方法内带出数组元素值。

实例 19：交换处理数组内的数据

下面通过一个简单实例来说明使用数组参数的具体流程，本实例保存在“光盘：\daima\7”文件夹内，项目名为 shuzucanshu。实例的功能是通过方法对数组内的数据进行交换处理。实例文件的主要代码如下所示：

```
namespace shuzucanshu{
    class Program
    {
        // 交换 a, b 的值,能成功
        public static void mm(int[] a, int[] b)
        {
            int c = a[0];
            a[0] = b[0];
            b[0] = c;
            Console.WriteLine("在方法 mm 内 a, b 的值");
            Console.WriteLine("a = " + a[0]);
            Console.WriteLine("b = " + b[0]);
        }
        // 交换 a, b 的值,不成功
        public static void mm2(int[] a, int[] b)
        {
            int[] c = new int[1];
            c = a;
            a = b;
            b = c;
            Console.WriteLine("在方法 mm2 内 a, b 的值");
            Console.WriteLine("a = " + a[0]);
            Console.WriteLine("b = " + b[0]);
        }
        // 交换 a, b 的值成功
        public static void mm3(ref int[] a, ref int[] b)
        {
            int[] c = new int[1];
            c = a;
            a = b;
            b = c;
            Console.WriteLine("在方法 mm3 内 a, b 的值");
            Console.WriteLine("a = " + a[0]);
            Console.WriteLine("b = " + b[0]);
        }
        static void Main(string[] args)
        {
            int[] a = { 3 };
            int[] b = { 50 };
            Console.WriteLine("交换前 a, b 的值");
            Console.WriteLine("a = " + a[0]);
            Console.WriteLine("b = " + b[0]);
            // 交换 a, b 的值
            mm(a, b);
            Console.WriteLine("交换后 a, b 的值");
            Console.WriteLine("a = " + a[0]);
```

```
                Console.WriteLine("b = " + b[0]);
                // 交换 a, b 的值
                mm2(a, b);
                Console.WriteLine("交换后 a, b 的值");
                Console.WriteLine("a = " + a[0]);
                Console.WriteLine("b = " + b[0]);
                // 交换 a, b 的值
                mm3(ref a, ref b);
                Console.WriteLine("交换后 a, b 的值");
                Console.WriteLine("a = " + a[0]);
                Console.WriteLine("b = " + b[0]);
                Console.ReadKey();
            }
        }
    }
```

上述实例代码的设计流程如下。

1）定义方法 mm()，实现变量数据 a 和 b 的交换处理，可以交换成功。

2）定义方法 mm2()，实现变量数据 a 和 b 的交换处理，不可以交换成功。

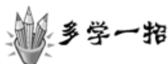
3）定义方法 mm3()，实现变量数据 a 和 b 的交换处理，可以交换成功。

4）分别调用各方法，将对变量 a 和 b 交换后的结果输出。

执行后将调用方法并输出对应的处理结果，效果如图 7−9 所示。

```
file:///E:/机械工业/c#/daima/7/shuzucanshu/bin/Debug/shuzucan
交换前a, b的值
a = 3
b = 50
在函数mm内a, b的值
a = 50
b = 3
交换后a, b的值
a = 50
b = 3
在函数mm2内a, b的值
a = 3
b = 50
交换后a, b的值
a = 50
b = 3
在函数mm3内a, b的值
a = 3
b = 50
交换后a, b的值
a = 3
b = 50
```

图 7−9　实例执行结果



多学一招

用 params 修饰符声明的参数是参数数组。不能将 params 修饰符与 ref 和 out 修饰符组合起来使用。

在一个方法的调用中，允许通过以下列两种方式之一来为参数数组指定对应的参数。

1）赋予参数数组的参数可以是一个表达式，它的类型可以隐式转换为该参数数组的类

型。在此情况下，参数数组的作用与值参数完全一样。

2）此调用可以为参数数组指定零个或多个参数，其中每个参数都是一个表达式，它的类型可隐式转换为该参数数组的元素的类型。在此情况下，此调用创建一个长度对应于参数个数、类型与该参数数组的类型相同的一个数组实例，并用给定的参数值初始化该数组实例的元素，并将新创建的数组实例用做实参。

7.5 结构方法

经过本书前面知识的学习，了解到在结构类型变量中可以存储多个数据。其实除了上述功能外，结构还可以包含方法和数据。看下面的结构格式：

```
struct mm
{
public string nn,zz;
}
```

在上述格式中，如果变量类型是 mm，并且需要在控制台上输出一个完整的名称，则必须从其组件部分建立该名称。例如，在 mm 变量 ff 中可以使用如下格式：

```
mm ff;
ff.nn=" woshi";
ff,zz=" shui";
Console.WriteLine("{0}{1}",ff.nn,ff.zz);
```

如果将方法放在某个结构中，就可以集中处理常见的任务，简化对应的过程。例如，下面的代码将方法添加到了结构类型中：

```
struct mm
{
public string nn,zz;
public string name()
     {
     return nn+""zz;
     }
}
```



方法 name()可以直接访问 nn 和 zz 内的结构成员。在上面代码中定义的结构 mm 内，可以将 nn 和 zz 看做全局成员。

7.6 方法重载

C#的方法签名是由方法的名称和它各形参类型与种类构成的。而在调用方法时，必须匹

配方法的签名。所以在代码编写时需要让多个方法操作不同类型的变量。为此，C#方法重载便提上了议程。

7.6.1　方法重载介绍

方法重载是指在同一个类中允许存在多个名称相同但签名不同的方法。签名是方法重载的基础，重载不能违背类内签名方法唯一性的特点。

因为重载使同一个类内有多个相同的方法，所以在编译一个重载方法的调用代码时，编译器使用重载决策来确定要调用的特定方法。通过重载决策查找与其参数最为匹配的方法，如果没有找到则返回错误信息。

在 C#中，要根据调用时的实参和方法声明中的形参的匹配程度来选择重载决策，具体说明如下。

1）调用时的实参数目和声明时的形参数目相同。

2）调用时的各实参参数、参数的传递模式，与相应的声明形参的相同。而且还应该具备如下两个特点：

1）对于值参数或参数数组，存在从实参类型向形参类型的隐式转换。

2）对于引用参数或输出参数，实参的类型和形参的类型相同。

在使用方法重载时，必须遵循上面的两个最佳原则。如果不存在最佳匹配方法，则在调用方法时会出现编译错误。



在使用重载方法时，需要读者注意如下三点。

1）方法的签名不包含返回类型。

2）方法的签名不包含修饰符 params，但是可以用于指定最右边的参数。

3）方法的签名可以包含修饰符 ref 和 out，但是不能区分出 ref 和 out。

下面通过一段代码看如何对 WriteLine()方法进行重载，具体代码如下：

```
namespace chongzai1
{
    class Program
    {
        static int ChongzaiHanshu(int x)
        {
            Console.WriteLine("开始使用函数 ChongzaiHanshu(int)");
            return x * x;
        }
        static double ChongzaiHanshu(double x)
        {
            Console.WriteLine("开始使用函数 ChongzaiHanshu(double)");
            return x * x;
        }
        static void ChongzaiHanshu(ref int x)
        {
```

```
            Console.WriteLine("开始使用函数 ChongzaiHanshu(ref int)");
            x *= x;
        }
        static int ChongzaiHanshu(int x, int y)
        {
            Console.WriteLine("开始使用函数 ChongzaiHanshu(int, int)");
            return x * y;
        }
        static void ChongzaiHanshu(ref int x, int y)
        {
            Console.WriteLine("开始使用函数 ChongzaiHanshu(ref int, int)");
            x *= y;
        }
        static void ChongzaiHanshu(int x, int y, out int z)
        {
            Console.WriteLine("开始使用函数 ChongzaiHanshu(int, int, out int)");
            z = x * y;
        }
        static void Main()
        {
            int i = 1, j = 2, k;
            ChongzaiHanshu(i);
            ChongzaiHanshu(ref i);
            ChongzaiHanshu(i, j);
            ChongzaiHanshu(ref i, j);
            ChongzaiHanshu(i, j, out k);
            double x = 2.45;
            ChongzaiHanshu(x);
            Console.ReadKey();
        }
    }
}
```

在上面的代码中，定义了多个名称相同但签名不同的方法 ChongzaiHanshu()。当程序运行后，将调用对应的重载方法，并输出显示对应的结果，具体如图 7-10 所示。

```
开始使用函数ChongzaiHanshu(int)
开始使用函数ChongzaiHanshu(ref int)
开始使用函数ChongzaiHanshu(int, int)
开始使用函数ChongzaiHanshu(ref int, int)
开始使用函数ChongzaiHanshu(int, int, out int)
开始使用函数ChongzaiHanshu(double)
```

图 7-10 实例执行结果

由图 7-10 所示结果可以看出，编译器能够根据不同参数的匹配程度正确地选择出最合适的调用方法。

7.6.2　方法重载的具体应用

重载是一种为方便代码编写而推出的处理机制，在使用重载时必须遵循签名唯一性的原则，否则应用程序将不能被正确的编译。看下面的一段代码：

```
class 类
{
  void mm()
   {
   …
}
void mm(int x)
   {
   …
}
void mm(ref int x)
   {
   …
}
  void mm(out int x)          //不能区分 ref 和 out
   {
   …
}
  void mm(int x,int y)
   {
   …
}
void mm(int x)                //int mm(int)和 void mm(int)仅仅是返回值不同
   {
   …
}

}
```

因为所有的 ref 修饰符和 out 修饰符都是签名的组成部分。所以方法 mm(int)和方法 mm(ref int)的签名具有唯一性。但是，方法 mm(ref int)和方法 mm(out int)不能在同一个类中声明，因为两者的签名仅仅是 ref 和 out 字符的不同。再看下面的一段代码：

```
namespace chongzai2
{
    class Chuli
    {
    private double[,] elements;
    private int row = 1;
    private int col = 1;
    public int Row
```

```
        {
            get
            {
                return row;
            }
        }
        public int Col
        {
            get
            {
                return col;
            }
        }
        public double[] this[int r]                             // 定义索引器
        {
            get                                                 // get 访问器
            {
                if (r >= row || r < 0)                          // 如果越界
                {
                    throw new IndexOutOfRangeException();       // 引发一个异常
                }
                double[] vector = new double[col];
                for (int i = 0; i < col; ++i)
                    vector[i] = elements[r, i];
                return vector;                                  // 取值
            }
            set                                                 // set 访问器
            {
                if (r >= row || r < 0)                          // 如果越界
                {
                    throw new IndexOutOfRangeException();       // 引发一个异常
                }
                for (int i = 0; i < col; ++i)                   // 赋值
                    elements[r, i] = value[i];
            }
        }
        public double this[int r, int c]                        // 重载索引器
        {
            get                                                 // get 访问器
            {
                if (r >= row || r < 0 ||
                    c >= col || c < 0)                          // 如果越界
                {
                    throw new IndexOutOfRangeException();       // 引发一个异常
                }
                return elements[r, c];                          // 取值
```

```
        }
        set                                                 // set 访问器
        {
            if (r >= row || r < 0 ||
                c >= col || c < 0)                          // 如果越界
            {
                throw new IndexOutOfRangeException();       // 引发一个异常
            }
            elements[r, c] = value;                         // 赋值
        }
    }
    public Chuli()
    {
        row = 1;
        col = 1;
        elements = new double[row, col];
    }
    public Chuli(int row, int col)
    {
        this.row = row;
        this.col = col;
        elements = new double[row, col];
    }
    public static void Main()
    {
        const int ROW = 8;
        const int COL = 8;                                  // 创建对象
        for (int i = 0; i < m.Row; ++i)                     // 赋值
        {
            for (int j = 0; j < m.Col; ++j)
            {
                m[i, j] = i * j;
            }
        }
        for (int i = 0; i < m.Row; ++i)                     // 打印
        {
            for (int j = 0; j < m.Col; ++j)
            {
                Console.Write("{0,4} ", m[i, j]);
            }
            Console.WriteLine();
             Console.ReadKey();
        }
        Console.WriteLine("\n 第五行的数字如下:");
        double[] vector = m[4];
        for (int j = 0; j < vector.Length; ++j)
```

```
                {
                    Console.Write("{0,4} ", vector[j]);
                }
                Console.ReadKey();
            }
        }
    }
```

上述实例代码的设计流程如下。

1）分别定义变量 Row 和 Col，用于返回矩阵的行数字和列数字。

2）定义数组 this，并返回行和列的数据。

3）重载定义数组 this，并返回行和列的数据。

4）设置行数和列数的值为 8。

5）通过 WriteLine()将数组内数据以矩阵样式排列。

6）通过重载方法将矩阵内第五行的数字输出。

上述代码保存在“光盘：\daima\7”文件夹内，项目名为 chongzai2。上述代码的功能是通过方法实现数字矩阵排列效果，并通过重载输出矩阵内第五行的数字。执行后将首先将数字内数据以矩阵样式排列，如图 7-11 所示；当按〈Enter〉键后将输出矩阵内第五行的数字，具体如图 7-12 所示。

```
   0    0    0    0    0    0    0    0
   0    1    2    3    4    5    6    7
   0    2    4    6    8   10   12   14
   0    3    6    9   12   15   18   21
   0    4    8   12   16   20   24   28
   0    5   10   15   20   25   30   35
   0    6   12   18   24   30   36   42
   0    7   14   21   28   35   42   49
```

图 7-11　实例执行结果

```
   0    0    0    0    0    0    0    0
   0    1    2    3    4    5    6    7
   0    2    4    6    8   10   12   14
   0    3    6    9   12   15   18   21
   0    4    8   12   16   20   24   28
   0    5   10   15   20   25   30   35
   0    6   12   18   24   30   36   42
   0    7   14   21   28   35   42   49

第五行的数字如下:
   0    4    8   12   16   20   24   28
```

图 7-12　输出矩阵内第五行的数字

7.7　使用 Main()方法

Main()方法是 C#程序的入口点，执行 Main()方法实际上就是执行应用程序。也就是说，在程序执行开始时会先执行 Main()方法，当 Main()方法执行结束后，执行程序的过程也就结束了。在本书前面的内容中，已经多次使用了 Main()方法。在本节的内容中，将对 Main()方法的基本知识进行详细介绍。

Main()方法的具体语法格式如下：

```
static 修饰符 Main()
```

Main()方法可以使用如下四种签名方式。

1）static void Main()。

2）static void Main(string[] args)。

3）static int Main()。

4）static int Main(string[] args)。



在使用 Visual Studio.NET 创建应用项目时，将自动生成一个默认格式的 Main()方法。例如，需要创建一个名为 mainmain 的项目，通过 Visual Studio.NET 处理后的默认代码如下:

```
namespace mainmain
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

Main()方法具有如下五个特点。

1）可以选择返回一个 int 类型值或不返回值。

2）应用程序可能由多个类组成，在这些类的结构中可能有多个 Main()方法。在此情况下，必须使用外部机制来设置一个 Main()方法作为程序的入口。

3）可以包含一个形参或没有参数，如果存在形参则参数的类型必须是 string[]。

4）C#支持重载，但必须具有一个以上的参数，或唯一的参数类型不是 string[]。但是这样的 Main()方法不能作为程序的入口。

5）应用程序从 Main()方法开始执行，当 Main()方法执行完毕后，应用程序将终止并控制返回给执行环境。

实例 20：使用 Main()方法

下面通过一个简单实例来说明 Main()方法的使用流程，实现代码保存在“光盘：\daima\7”文件夹内，项目名为 main。实例的功能是调用 Main()方法来传递命令行参数。本实例的实现流程如下。

1）定义入口方法 main()。

2）通过 if 语句对命令行参数进行判断处理。

3）如果命令行参数为空则输出“how are you！”。

4）如果命令行参数不为空，则输出指定格式的文本。

上述流程的具体实现过程如下。

1．编写 CS 文件代码

实例文件 Program.cs 的主要代码如下：

```
namespace main
{
    class Program
    {
```

```
            public static void Main(string[] args)
            {
                if (args.Length == 0)
                {
                    Console.WriteLine("how are you!");
                    Console.ReadKey();
                }
                else
                {
                    for (int i = 0; i < args.Length; ++i)
                        Console.WriteLine("how are you {0}!!!! ", args[i]);
                        Console.ReadKey();
                }
            }
        }
    }
```

上述实例代码执行后，因为没有任何命令行参数，所以将输出文本“how are you!”，具体如图 7-13 所示。

file:///F:/机械视频/C#/daima/

how are you!

图 7-13　实例执行结果

2．设置命令行参数

设置命令行参数的操作步骤如下。

1）在 Visual Studio 2010 中打开上面的项目 main，如图 7-14 所示。

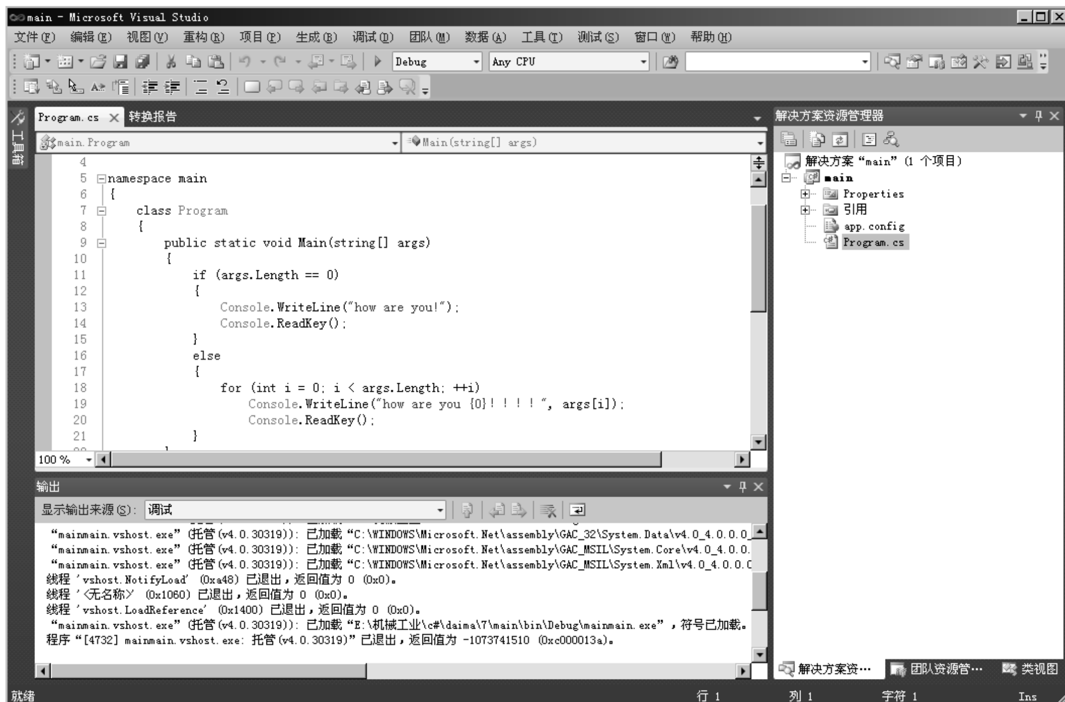


图 7-14　Visual Studio.NET 界面

2）在右侧“解决方案”选项中双击“Properties”选项，弹出“属性配置”对话框，如图 7-15 所示。

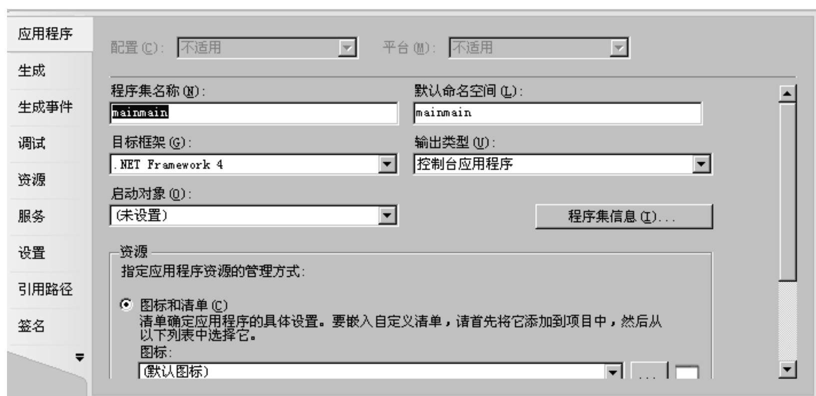


图 7-15　“属性配置”对话框

3）单击右侧项目列表中的“调试”选项，并在“启动项目”内 “命令行参数”后的文本框内分别输入 mm、nn、zz 和 ff 四个参数，如图 7-16 所示。



图 7-16　输入参数

经过上述操作处理后，执行项目后将显示指定格式的文本，如图 7-17 所示。

```
file:///E:/机械工业/c#/daima/7/main/bin/Debug/
how are you mm! ! ! !
how are you nn! ! ! !
how are you zz! ! ! !
how are you ff! ! ! !
```

图 7-17　实例执行结果


多学一招

在通过 Main()方法传递了命令行参数时，Main()方法可以使用如下四种签名方式。

1）static void Main()。

2）static void Main(string[] args)。

3）static int Main()。

4）static int Main(string[] args)。

并且在使用 Visual Studio.NET 创建应用项目时，将自动生成一个默认格式的 Main()方法。例如，需要创建一个名为 mainmain 的项目，通过 Visual Studio2010 处理后的默认代码如下:

```
namespace mainmain
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

7.8 递归

递归是常用的编程方法之一，通过递归可以在一个代码块中直接或间接地调用自己。C#语言支持递归，具体来说分为如下两类。

1）直接递归：在方法中调用方法本身。

2）间接递归：在某方法内调用另一个方法，而在另一个方法内调用某方法。

看下面的一段代码：

```
class Program
    {
        public static uint Chuli(uint mm)
        {
            uint ff = 1;
            for (uint i = 1; i <= mm; ++i)
            {
                ff *= i;
            }
            return ff;
        }
    }
```

在上述代码中，通过定义的方法 Chuli()实现了数据的阶乘 mm!处理。具体说明如下。

1）如果变量 mm 值为 1，则返回 1。

2）如果变量 mm 的值大于 1，则返回 mm!。

如果使用递归方法，则可以对上述代码进行如下修改：

```
        class Program
            {
                public static uint Chuli(uint mm)
                {
                    uint ff = 1;
                    for (uint i = 1; i <= mm; ++i)
                    {
                        ff *= i;
                    }
                    return ff;
                }
                public static uint Chuli1(uint mm)
                {
                    if (mm==1)
                    {
                        return 1;
                    }
                    else
                    {
                        return mm*Chuli1(mm-1);
                    }
                }
            }
```

在上修改后的代码中，如果变量 mm 的值不是 1，则递归调用方法 Chuli1()来计算结成的值。

任何递归都必须满足能达到的某个条件，如果满足这个条件则从方法中返回结果。这个条件出口通常被称为递归出口。如果没有递归出口，则可能会使整个系统崩溃。

在 C#内，能够使用循环实现的功能都能使用递归来代替实现。

7.9　疑难问题解析

本章详细介绍了 C#方法的基本知识。本节将对本章中比较难以理解的问题进行讲解。

读者疑问：C#的方法参数有具体要注意的吗？

解答：在 C#方法中如果有多个参数，可以使用逗号“,”进行分割。并且该参数和方法定义中的方法类型所对应，或者可以隐式的转换为那种类型。

在有特殊需要时，可以使用方法的完全限定名，即包含具体的命名空间和类。例如前面实例中常用的“System.Console.WriteLine”。

读者疑问：C++中有方法重载吗，是否和 C#的原理一致？

解答：以前在 C++中曾经知道 C++中有方法重载的概念，现在在 ASP.NET 中仍然可以使用方法重载的概念和定义。假设在程序中定义了两个方法 String test(String str1) 和 int test

(int i)，它们的内容都是很简单的功能：

```
String test(String str1){
Response.Write("方法重载测试，这个是String方法");
Return "123232";
}
int test(int i){
Response.Write("方法重载测试，这个是int方法");
Return 1;
}
```

当调用“豆腐”和“888”，编译器非但不会报错，而且会执行出不同的结果。这样，方法重载在编程过程中带来了很大的方便，可以用编译器来决定根据参数变化执行的不同代码。而且这也是面向对象编程的一个主要的特色之一。相信微软公司的确想让 ASP.NET 成为 Web 未来编程的规范。



职场点拨——模块化设计的原则

在设计大型程序时，常常要将整个问题分解为若干个小问题，必要时还要将小问题再次分解为更小的若干问题，每个小问题编写成独立的源文件，最后将所有的源文件连接起来组合成一个大程序。也就是说，一个程序往往由多个源文件组成，构成一个程序的各个相对独立的源文件通常称为模块。这样把一个程序分成多个功能相对独立的程序模块分别编制、调试后，再用连接程序把它们连接在一起生成一个完整的程序的设计的方法称为模块化程序设计。

在当前市面上有很多模块方面的书籍，里面罗列了开发中的常用模块，这些模块对开发人员来说很重要。模块是指程序中的一段代码，该段代码能够实现程序中的某一功能并能独立或半独立运行。在大型程序编写中，模块化的运用是不可避免的。从面向对象编程思想被推出后，模块化程序设计思想就成为了一个主流的软件开发模式。

既然模块化设计这么重要，在此简要介绍模块化设计的基本原则，希望读者好好把握。

1）一个主模块完成对各子模块的调用，实现总体任务，而每个子模块完成相应的子任务，各模块间除应在功能上分开，逻辑上独立，减少横向联系外，不能使用转移指令在模块间转来转去，避免逻辑上的混乱。

2）子模块大小应适中，模块过大就失去了模块化的意义，也会给编程和调试带来一定困难；模块过小，会在时间和空间上造成浪费。

3）差别很大的两个程序段应作为两个模块。

4）当一些数据被多个程序段所公用，那么这些数据所在的程序段应作为一个模块。

5）当某些程序功能片段为多个模块所公用时，应将它们作为公用子程序模块。

6）各个模块的结构最好能设计为单入口、单出口的形式，各模块间的接口应该简单，要尽量减少公共标识符的个数。

第 8 章　类

类是 C#语言中最核心的内容之一，它表示了一种数据结构，能够封装数据成员、函数成员和其他的类。类是 C#语言的基础，可以将 C#内的一切类型看做类，并且所有的语句都位于类内。另外，C#还支持自定义类，用户可以根据需要在程序内定义自己需要的类。在本书前面的内容中，已经使用了大量的类，例如 Program。在本章的内容中，将详细介绍 C#类的基本知识，并通过简单的实例来介绍其具体使用流程。通过本章能学到如下知识。

- ❑ 定义类和类的成员。
- ❑ 访问修饰符。
- ❑ C#类成员。
- ❑ 使用构造函数和析构函数。
- ❑ 只读字段和 this 关键字。
- ❑ C#属性和索引器。
- ❑ 命名空间。
- ❑ C#内部类和分布类。
- ❑ 职场点拨——谈职业精神。

2007 年 XX 月 X 日，天气阴

今天，我从 HR 那里听说了一个新名词——职业精神，并且 HR 要求我们继续提高我们的职业精神。为了弄清楚什么是职业精神，我特地百度了一下：“职业精神是与人们的职业活动紧密联系、具有自身职业特征的精神”。



一问一答

小菜：职业精神真的那么重要吗？难道技术不是最重要的？

Wisdom：职业精神是一个职场人是否能最终成功的关键，即使技术功底深厚，但如果没有职业精神，最终也会走向失败。你应该看过武侠小说吧，如果小说中的坏人武功越高，破坏力就越大！在本章的最后，将从沟通着手谈谈职业精神。

小菜：嗯，言归正传，本章将要学的类很重要吗？

Wisdom：在 C#中，所有的类型都是基于类的。因为有了面向对象思想，所以在高级语言中推出了类这一概念。类的地位犹如职业精神，是程序是否健壮、科学的根本。

8.1 定义类

在 C#中，类使用关键字 class 来定义。只有经过定义声明后，才能在应用程序中使用类。可以使用诸如 int 和 double 之类的基本类型来对类进行修饰。

定义 C#类的格式如下：

```
修饰符 class 类名
{
  类成员
}
```

在定义类时，可以使用修饰符来设置类的作用范围。使用关键字 internal 修饰的类是内部类，即只能在当前项目中访问使用。例如下面的类 class1 就是一个内部类：

```
internal class class1 {
  类成员
}
```

如果要想使定义的类能够在多个项目中使用，需要使用关键字 public 来设置为公共类。例如下面的类 class1 就是一个公共类：

```
public class class1 {
  类成员
}
```

另外，还可以使用使用抽象和封装关键字来对类进行修饰。例如，下面的代码使用了抽象关键字 abstract 声明一个抽象类 class1：

```
public abstract class class1 {
  类成员
}
```

而下面的代码使用了封装关键字 sealed 声明一个抽象类 class1：

```
public sealed class class1 {
  类成员
}
```

并且抽象类和封装类也可以被定义为内部类或公共类。例如，在下面的代码中分别定义了内部抽象类 class1 和内部封装类 class2：

```
internal abstract class class1 {
  类成员
}
internal sealed class class2 {
  类成员
}
```

如果一个类 a 是在另一个类 b 中定义声明的，则通常将类 b 作为类 a 的基类，而类 a 是类 b 的派生类。在定义类的同时，可以指明它所属的基类。例如，下面代码中的类 mm 就是类 nn 的基类：

```
public class mm:nn{
  类成员
}
```



在 C#程序中，每个类只能指明一个基类，并且不允许派生类的可访问性高于基类的可访问性。即内部类可以继承于一个公共类，但公共类不能继承于一个内部类。例如下面的代码是非法的：

```
internal class class1{
  类成员
}
public class class2: class1
{
  类成员
}
```

8.2　类的成员

当定义声明一个 C#类后，在类体内的所有元素都是这个类的成员。C#的类成员有两种，分别是数据成员和函数成员。在本节的内容中，将对上述两种类成员的基本知识进行简要介绍。

8.2.1　最简单的数据成员

C#的数据成员包括字段和常量两种，具体说明如下。

1．字段

字段是在类中定义的成员变量，能够存储描述这个类的特征值。在类内的字段可以预先初始化声明，声明的字段将作用于整个类体。例如，在下面的代码中分别定义了两个不同类型的字段 mm 和 nn：

```
class lei
  {
    string mm= " aaa ";
    int nn=20;
  }
```

2．常量

常量是在类内定义的常量成员，在本书第 3 章中介绍的常量声明方法也适用于类内的常量成员。例如，在下面的代码中定义了常量 mm：

```
class lei{
    const int mm=20;
    string mm= " aaa ";
    int nn=20;
}
```

8.2.2 最重要的函数成员

C#类的函数成员主要包括六种，分别是函数、属性、索引器、事件、运算符、构造函数和析构函数。在下面的内容中，将分别介绍上述六种函数成员。

1．方法

方法即函数，是用来实现某特定功能的计算和操作，在 C#的类中可以定义和调用需要的方法。例如在下面的代码中，在类 chuli 内定义了方法 mm()：

```
class Program{
     public static uint Chuli(uint mm)
     {
       uint ff = 1;
       for (uint i = 1; i <= mm; ++i)
       {
          ff *= i;
       }
       return ff;
     }
}
```

2．属性

属性是字段的扩展，并且属性和字段都是命名的成员，都有对应的类型，访问两者的语法格式也相同。两者唯一的区别是属性不能表示存储位置，并且属性有访问器。

3．索引

索引和属性基本类似，但是索引能够使类的实例按照和数组相同的语法格式进行检索。

4．事件

事件常用于定义可以由类生成的通知或信息，通过事件可以使相关的代码激活执行。

5．运算符

运算符常用于定义对当前类的实例进行运算处理的运算符，可以对预定义的运算符进行重载处理。

6．构造函数和析构函数

名称和类相同的函数叫做构造函数，当类被实例化后，首先被执行的就是构造函数。另外，析构函数也是一种特殊的函数，其名称是和在类名前加字符“~”。当前类无效时，会执行定义的析构函数。



注意

在上述函数成员中，除了方法在前面的章节内进行了详细介绍外，只是简单介绍了其他的成员，甚至有的根本没有介绍。读者只要了解上述函数成员即可，至于其他成员的具体信息将在本书后面的章节中进行详细介绍。

8.3 访问修饰符

访问修饰符能够控制修饰成员的可访问范围。访问修饰符能够使类或类的成员在不同的范围内具有不同的可见性，用于实现数据和代码的隐藏。在 C#中定义了五种访问修饰符，分别是 public、protected internal、protected、internal 或 private。在上述修饰符中，除了 protected internal 组合外，其他的访问修饰符组合方式会导致编译时的错误。访问修饰符被设置在成员的类型或返回值类型之前。

C#常用访问修饰符的具体说明如表 8-1 所示。

表 8-1 C#访问修饰符

访问修饰符	说 明
public	设置访问不受限制，可以在类内和任何类外的代码中访问
protected	设置可访问域限定于类内或从该类派生的类内
internal	设置可访问域限定于类所在的程序内
protected internal	即表示 protected 或者 internal，设置可访问域限定于类所在的程序或那些由它所属的类派生的类内
private	设置可访问域限定于它所属的类内

8.3.1 类成员访问修饰符

前面介绍的五种访问修饰符都可用于修饰类内的成员。在没有定义任何访问修饰符时，类成员的默认访问修饰符是 private。看下面的一段代码：

```
class aa{
   public const int RETIREMENT_AGE =50;
   public string name;
   protected bool isMarried;
   private int age;
   string zzzzz;
   public void Speak()
   {
     Console.WriteLine("你好!");
  }
   private void WriteDiary() {
     Console.WriteLine("你好呀! ");
  }
}
class bb : aa{
…
}
class cc{
…
}
```

在上述代码中，共同存在着 aa、bb 和 cc 三个类，其中类 bb 是类 aa 的派生类。根据上

述修饰符的设置，类 aa 内成员可访问性的具体说明如下。

- RETIREMENT_AGE，访问不受限制。
- name，访问不受限制。
- isMarried，只有在类 aa 或者其派生类中可访问。
- age，只能在类 aa 内可访问。
- zzzzz，使用了默认访问修饰符 private，只在类 aa 内可访问。
- Speak()，访问不受任何限制。
- WriteDiary()，只能在类 aa 内可以访问。

在上述代码中，如果在类 bb 内调用类 aa 的成员，则应该按照修饰符定义的可访问性来确定是否合法。类 bb 内 aa 的成员的可访问性信息如下。

- RETIREMENT_AGE，可以访问。
- name，可以访问。
- nickName，可以访问。
- isMarried，可以访问。
- age，不可以访问。
- zzzzz，不可以访问。
- Speak，可以访问。
- WriteDiary，不可以访问。



在上述代码的类 bb 内，成员的可访问性遵循了对应修饰符的范围。即 public、protected 和 internal 修饰成员都可访问，而 private 修饰成员不可访问。

在上述代码中，如果在类 cc 内调用类 aa 的成员，则对应的成员可访问性信息如下。

- RETIREMENT_AGE，可以访问。
- name，可以访问。
- nickName，可以访问。
- isMarried，不可以访问。
- age，不可以访问。
- zzzzz，不可以访问。
- Speak，可以访问。
- WriteDiary，不可以访问。



因为程序内的类 cc 不是类 aa 的派生类，所以在类 cc 内，public 和 internal 的修饰符成员都可以访问，而 protected 和 private 的修饰符成员不可以访问。

8.3.2 类的访问修饰符

在 C#中，将不在类中声明的类称做顶级类。在应用程序中，顶级类只能使用 public 和 internal 两种访问修饰符。这两种修饰符的具体说明如下。

1）修饰符 public，被修饰顶级类的可访问范围是它所在的程序和任何引用该程序的程

序，即访问不受限制。

2）修饰符 internal，被修饰的顶级类的可访问范围是在定义它的程序内。

如果在声明顶级类时没有指定访问修饰符，则默认的访问修饰符是 internal，表示此类只能被定义它的程序所使用。看下面的一段代码：

```
public class xx {
…
}
internal class yy {
  …
}
class zz{
  …
}
```

在上述代码中，分别定义了 xx、yy 和 zz 三个顶级类，各类的可访问性的具体说明如下。

- 类 xx：因为是用 public 修饰的，所以可以被任何引用它所在的程序的程序访问。
- 类 yy：因为是用 internal 修饰的，所以只能在它所在的程序内访问。
- 类 zz：因为没有声明访问修饰符的类，所以要使用默认的修饰符 internal，所以只能在它所在的程序内访问。

8.4　对象是什么

C#通过类的定义实现了模块化处理和面向对象编程，难道仅仅使用类的初始状态就能进行特定操作吗？答案是否定的，还必须将类进行对象处理后才能实现具体的功能。在本节的内容中，将对 C#对象的基本知识进行详细介绍，并通过具体实例的实现过程来介绍其使用流程。

8.4.1　创建对象

C#的类是抽象的，将类实例化后才能实现特定的功能，即创建类的对象后才能使用。类与对象的关系可以比喻为机型设计和具体的机器，类是一系列属性和功能的描述，而对象就像根据类的描述而设计出的具体处理程序。在 C#中为类创建的对象的具体格式如下：

```
类名 对象名= new 类名 (参数);
```

其中，参数是可选的。

例如，在下面的代码中为 mm 类创建了一个对象 nn：

```
mm nn = new mm();
```



在 C#编程语言中，创建类的对象、创建类的实例和实例化类等说法是相同的，都是以类为模板生成一个对象的操作。

8.4.2 使用对象

在 C#程序中，对象使用点运算符“.”来引用类的成员，并且引用的范围受到成员的访问修饰符的控制。看下面的一段代码：

```
class mm{
   public string aa;
   protected string bb;
   private string cc;
   internal string dd;
   public void Dive() {
    }
}
class ff{
   public static void Main()
   {
      mm nn = new mm();
      nn.aa = "abc123";
      nn.bb = "aaa";
      nn.cc= "aa";                    // 错误：在类外不能引用 private 成员
      nn.dd = "cv ";                  // 在程序内能引用 internal 成员
      nn.Drive();                     // 能引用 public 成员
}
}
```

在上述代码中，在类 ff 内通过点运算符的方式引用了类 mm 内的成员。例如 nn.aa、nn.bb 和 nn.cc。根据访问修饰符的控制范围，得出类 mm 的对象 nn 不能在 mm 类外引用类的 protected 和 private 成员。所以在上述代码中，对象 nn 的运行权限结果如下。

- nn.aa：可以引用，因为可以引用 public 成员。
- nn.bb：不可以引用，因为在类外不可以引用 protected 成员。
- nn.cc：可以引用，因为在程序内可以引用 internal 成员。
- nn.Drive()：可以引用，因为可以引用 public 成员。


注意

在使用关键字 new 创建一个类的对象时，将在托管堆中为对象分配一块内存，每一个对象都有不同的内存。代表对象的变量存储的是存放对象的内存的地址。因此，两个不同的对象，即使它们的所有成员的值或代码都相同，它们也是不相等的。

实例 21：根据用户姓名输出对应 QQ 名

下面通过一个简单实例来说明使用对象的流程，本实例保存在光盘：\daima\“8”文件夹内，项目名为 duixiangshiyong。实例的功能是根据用户的姓名输出对应的 QQ 名。实例文件的主要代码如下：

```
namespace duixiangshiyong {
   class QQming
```

```
        {
            public string name;
            public void Bark()
            {
                Console.WriteLine("大爱无敌!");
            }
        }
        class DQQming
        {
            public static void Main()
            {
                QQming mm = new QQming();
                mm.name = "AA";
                QQming nn = new QQming();
                nn.name = "BB";

                Console.WriteLine("对象 mm 的名字为“{0}”，QQ 名是:", mm.name);
                mm.Bark();
                Console.WriteLine("对象 nn 的名字为“{0}”，QQ 名是:", nn.name);
                nn.Bark();

                if (mm == nn)
                {
                    Console.WriteLine("—> mm 与 nn 是同一个。");
                    Console.ReadKey();
                }
                else
                {
                    Console.WriteLine("—> mm 与 nn 不是同一个。");
                    Console.ReadKey();
                }
            }
        }
    }
```

上述实例代码的设计流程如下。

1）分别定义类 QQming 和 DQQming。

2）在类 QQming 内定义函数 Bark()，用于输出用户的 QQ 名。

3）在类 QQming 内创建类 QQming 的对象，并分别调用 QQming 的成员。

4）分别设置两个用户名，并将对应的名字和 QQ 名通过 WriteLine()输出。

5）根据用户名判断是否为同一个用户对象，并将判断结果输出。

执行后将显示对应的用户名和 QQ 名，并显示是否为同一个用户对象的判断结果，如图 8-1 所示。

```
file:///E:/机械工业/c#/daima/8/duixiangshiyo
对象mm的名字为“AA ”，QQ名是:
大爱无敌!
对象nn的名字为“BB ”，QQ名是:
大爱无敌!
—> mm与nn不是同一个。
```

图 8-1　实例执行结果

在上述实例代码中，虽然两用户的 QQ 名相同，都是“大爱无敌”，但是他们并不是同一个人。



多学一招

在 C#对象中，如果将一个对象赋值给另一个对象，那么这两个对象就是相同的，代表它们的变量都将保存同一块内存的地址中。如果改变其中一个对象内成员的状态，则也会影响另一个对象内的成员的状态。看下面的一段的代码:

```
namespace duixiang1{
   class mm {
      public string name;
      public void Bark()
      {
        Console.WriteLine("寂寞天使!");
      }
   }
   class nn{
      public static void Main(){
        mm aa = new mm();
        aa.name = "张三";
        mm dd = new mm();
        dd = aa;
        if (aa == dd) {
           Console.WriteLine("aa 与 bb 是同一个人。");
        }
        else {
           Console.WriteLine("aa 与 bb 不是同一个人。");
        }
        Console.WriteLine("aa 的名字为“{0}”, QQ 名为:", aa.name);
        aa.Bark();
        Console.WriteLine("dd 的名字为“{0}”, QQ 名为:", dd.name);
        dd.Bark();
        dd.name = "李四";
        Console.WriteLine("改变 dd 的名字为“{0}”", dd.name);
        Console.WriteLine("aa 的名字也为会改变为“{0}”", aa.name);
        Console.ReadKey();
      }
   }
}
```

上述实例代码的设计流程如下。

1）分别定义类 mm 和 nn。

2）在类 mm 内定义函数 Bark()，用于输出用户的 QQ 名。

3）在类 nn 内创建类 mm 的对象，并分别调用 mm 的成员。

4）分别设置两个用户名，并将对应的名字和 QQ 名通过 WriteLine()输出。

5）根据用户名判断是否为同一个用户对象，并将判断结果输出。

执行后将显示对应的用户名和 QQ 名，并显示用户名修改后的处理结果，执行结果如图 8-2 所示。

从图 8-2 的执行结果可以看出，如果修改了对象 dd 的值，对应的对象 aa 的值也会修改。这是因为 aa 和 dd 是同一个用户。

```
aa与bb是同一个人。
aa的名字为“张三”，QQ名为：
寂寞天使!
dd的名字为“张三”，QQ名为：
寂寞天使!
改变dd的名字为“李四”
aa的名字也为会改变为“李四”
```

图 8-2　实例执行结果

8.5　C#类成员

C#类内的成员分为静态成员和实例成员两种。在本节的内容中，将详细讲解上述 C#类成员的基本知识，并通过具体实例的实现过程来介绍其使用流程。

8.5.1　类成员概述

在类成员的类型或者返回值类型前面添加关键字 static，就能使该成员变为静态成员。C#的常量或类型声明可以隐式的声明为静态成员，而其他没有用 static 修饰的成员被称为实例成员或者称为非静态成员。

C#中静态成员属于类，能够被这个类的所有实例所共享；实例成员属于对象，C#内的每一个对象都有实例成员的不同副本。C#的静态成员具有如下三个特点。

1）静态成员必须通过类名使用点运算符“.”来引用，而不能用对象来引用。

2）一个静态字段只标识一个存储位置。无论创建了一个类的多少个实例，它的静态字段在内存中都只占同一块区域。

3）静态函数成员不能作用于具体的实例，在这类函数成员中不能直接使用实例成员，必须通过类名来引用。例如函数方法、属性、事件、运算符或构造函数。

实例成员具有如下三个特点。

1）实例成员必须通过对象名并使用点运算符“.”来引用，而不能用类名来引用。

2）类的实例字段属于类的实例所有，当创建一个类的实例时，都在内存中为实例字段新建一个特定区域。类的每个实例分别包含一组该类的所有实例字段的副本。

3）实例函数成员作用于类的给定实例，在它们的代码体内可以直接引用类的静态和实例成员。例如函数、属性、索引器、实例构造函数或析构函数。

在本书前面的实例中，经常使用的 WriteLine()等方法就是典型的静态方法，都是通过类名 Console 加点运算符“.”来引用的。

8.5.2　类成员使用实例

在前面的内容中，了解了类成员的基本特点和应用方式。在接下来的内容中，将通过一个具体的实例的实现过程，来讲解程序应用类静态成员和实例成员的方法。

实例 22：输出显示类中的字段

下面通过一个简单实例来说明使用类成员的流程，实例代码保存在“光盘：\daima\8”文件夹内，项目名为 Leichengyuan。实例的功能是对类中的字段进行赋值，并将处理后的结果输出显示出来。实例文件的主要代码如下所示：

```
namespace Lei chengyuan{
   public class mm {
      static int i;
      int number;
      public mm()
      {
        i = i + 1;
        number = i;
      }
      public void show()
      {
        Console.WriteLine("对象{0} : count={1}", number, i);
      }
   }
   class nn
   {
      public static void Main()
      {
        mm a = new mm();
        a.show();
        Console.WriteLine("\n");
        mm b = new mm();
        a.show();
        b.show();
        Console.WriteLine("\n");
        mm c = new mm();
        a.show();
        b.show();
        c.show();
        Console.ReadKey();
      }
   }
}
```

上述实例代码的设计流程如下。

1）分别定义类 mm 和 nn。

2）在类 mm 内定义静态字段 i 和实例字段 number。

3）通过实例函数 show()将赋值后的 i 输出。

4）在类 nn 内分别创建对象 a 和 b。

5）创建的对象 a 和 b 分别调用函数 show()，输出显示对应的 i 值。

执行后将显示对应实例静态字段 i 的值，具体如图 8-3 所示。

```
file:///E:/机械工业/c#/daima/8/Leichengy
对象1 : count=1

对象1 : count=2
对象2 : count=2

对象1 : count=3
对象2 : count=3
对象3 : count=3
```

图 8-3　实例执行结果

8.6 使用构造函数

构造函数的功能是初始化程序内类的实例，这是 C#类的特殊函数之一。在本节的内容中，将对 C#构造函数的基本知识进行简要介绍，并通过具体实例的实现来讲解其使用流程。

8.6.1 构造函数概述

C#构造函数能够初始化类的实例。在 C#程序中，每个类都有构造函数，即使没有为类声明构造函数，编译器也会自动为类提供一个默认的构造函数。

构造函数的名称与所属类的名称相同，其基本特点如下。

- ❑ 构造函数不声明返回类型，也没有返回值。
- ❑ 默认的构造函数是 public 类型，如果是 private 类型的构造函数，则说明此类不能被实例化，通常用于只含有静态成员的类。
- ❑ 在构造函数中不要做对类的实例进行初始化以外的事情，也不能被显式的调用。

实例 23：用构造函数输出指定的文本

下面通过一个简单实例来说明使用 C#构造函数的流程，实例代码保存在“光盘：\daima\8”文件夹内，项目名为 gouzaohanshu。实例的功能是在类中声明一个构造函数，并调用函数输出指定的文本。实例文件的主要代码如下：

```
namespace gouzaohanshu {
    class mm
    {
    public string name;
    public mm()
    {
        name = "AA";
        Console.WriteLine("构造函数 mm()：类 mm 已被初始化。");
    }
    public void Bark()
    {
        Console.WriteLine("BB!");
    }
    public static void Main()
    {
        mm nn = new mm();
        Console.WriteLine("对象 nn 的名字“{0}”，\n 网名:", nn.name);
        nn.Bark();
        Console.ReadKey();
    }
}
}
```

上述实例代码的设计流程如下。

1）定义类 mm，并在其中声明一个构造函数 mm()。

2）通过 WriteLine()输出指定文本。

3）定义函数 Bark()，输出指定的网名。

4）在类 mm 内创建实例对象 nn，并输出指定的用户名和网名。

5）创建的对象 a 和 b 分别调用函数 show()，输出显示对应的 i 值。

上述实例代码执行后，将显示构造函数的执行过程，并显示对应的用户名和网名，如图 8-4 所示。

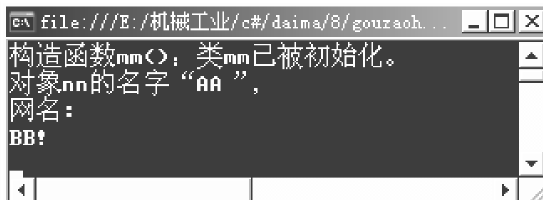


图 8-4　实例执行结果

从图 8-4 的执行效果可以看出，在访问一个类时将首先执行构造函数中的语句。构造函数的功能是创建对象，使对象的状态合法化。在从构造函数返回之前，对象都是不确定的，不能用于执行任何操作。只有在构造函数执行完成之后，存放对象的内存块中才存放这一个类的实例。很显然，在创建对象时执行了构造函数。


多学一招

在创建程序内类的一个实例时，在执行构造函数之前，系统会给所有未初始化字段设置一个默认值，然后以文本顺序执行各个实例字段的初始化。各类型字段的默认值信息如表 8-2 所示。

表 8-2　各类型字段的默认值

类　型	默 认 值
数值型	0
char	'\0'
bool	false
枚举	0
引用类型	null

8.6.2　构造函数的参数

在 C#类中可以同时存在多个具有不同参数的构造函数。在前面 8.6.1 小节中介绍的应用实例是不带参数的，这样的构造函数将以固定的方式来初始化类的实例。如果构造函数存在参数，这样的构造函数就能传递不同的数据，来实现对类的实例进行不同的初始化处理。

8.6.3　使用私有构造函数

使用关键字 private 声明的构造函数被称为私有构造函数。私有构造函数既不能从类外

访问，也不能用来在类外创建对象。

私有构造函数一般仅应用于只包含静态成员的类。这样的类不需要实例化，就能通过类名来引用所有的静态成员。在具体应用中，因为通过对象后会造成不能引用它的静态成员，因此需要有意使它不能被实例化。只需给这样的类添加一个空的私有实例构造函数，即可达到这一目的。例如下面的代码：

```
class aa {
  private aa() {}
  public const double a = 3.1;
  public static double Sin(double a) {...}
  public static double Cos(double a) {...}
  }
```

在上述代码中，通过 private 声明了一个私有构造函数 aa()，阻止被实例化的处理。

8.6.4　使用静态构造函数

在 C#中，使用关键字 static 声明的构造函数被称为静态构造函数。静态构造函数能够实现初始化类所需的操作，而不是初始化实例或对象。

C#类的静态构造函数在给定程序中至多只能执行一次，在程序中第一次执行某事件时，将触发静态构造函数的执行。具体的对应事件如下。

1）创建类的实例。

2）引用类的任何静态成员。

当初始化一个类时，在执行静态构造函数之前，首先将该类中的所有静态字段初始化为它们的默认值，具体如表 8-2 所示，然后以文本顺序执行各个静态字段的初始化。

如果类中包含用 Main()方法，那么该类的静态构造函数将在调用 Main()方法之前执行。如果类包含任何在声明时初始化的静态字段，则在执行该类的静态构造函数时，先要按照文本顺序执行那些初始化。

实例 24：用静态构造函数输出指定的文本

下面通过一个具体的实例来说明使用静态构造函数的流程。本实例保存在“光盘：daima\8”文件夹内，项目名为 jingtai。实例的功能是通过调用静态构造函数输出指定的文本。实例文件的主要代码如下：

```
namespace jingtai
{
   class mm
   {
      static void Main()
      {
        A.F();
        B.F();
      }
   }
   class A
```

```
        {
           static A()
           {
             Console.WriteLine("执行 A 的静态构造函数来初始化 A");
           }
           public static void F()
           {
             Console.WriteLine("调用 A.F()");
           }
        }
        class B
        {
           static B()
           {
             Console.WriteLine("执行 B 的静态构造函数来初始化 B");
           }
           public static void F()
           {
             Console.WriteLine("调用 B.F()");
             Console.ReadKey();
           }
        }
     }
```

上述实例代码的设计流程如下。

1）分别定义类 mm、A 和 B。

2）在类 A 内定义静态构造函数 A()，在类 B 内定义静态构造函数 B()。

3）调用创建的构造函数，输出对应的结果。

执行后将输出显示默认的值，具体如图 8-5 所示。

```
执行A的静态构造函数来初始化A
调用A.F()
执行B的静态构造函数来初始化B
调用B.F()
```

图 8-5　实例执行结果

从图 8-5 所示的执行结果可以看出，因为 A 的静态构造函数的执行是通过调用 A.F()触发的，而 B 的静态构造函数的执行是通过调用 B.F()触发的。所以首先执行 A，然后再执行 B。

多学一招

静态构造函数是非实例调用的，而非静态构造函数需要通过一个实例才能调用，所以在此要严加区别。如果在一个基类中它的继承类有一个静态的构造函数，在实例化继承类的时候，会首先调用继承类的构造函数，然后调用基类的。如果是构造函数，首先调用基类的构造函数,然后再调用继承类的构造函数。

如果要在继承类的静态构造函数中首先调用基类的，可以用一个 base 关键字来调用。读者可以查阅更多构造函数的相关知识，并尝试对上述实例进行修改，想办法不通过静态的构造函数实现上述功能，还可以对上述实例中的文本进行修改，以设置输出其他指定的文本。

8.7　使用析构函数

析构函数能够销毁类内的指定实例，并释放这个实例所占用的内存。析构函数是 C#类的特殊函数之一，在类名前面加上“~”字符即可构成析构函数的名称，具体格式如下：

```
~类名
```

析构函数具有如下三个特征。

- 没有返回值类型，也没有返回值。
- 没有参数。
- 一个类只能有一个析构函数。

C#的析构函数不能被显式调用，它是自动调用的。当程序内某个类的实例被认为不再有效、任何代码都不再使用它时，该实例就符合被销毁的条件。这样，它所对应的实例的析构函数就随时可能被调用，但是调用时机由公共语言运行时的垃圾回收机制确定，而不是由用户程序确定。

实例 25：用析构函数输出指定的文本

下面通过一个具体的实例来说明使用析构函数的流程。本实例保存在“光盘：\daima\8”文件夹内，项目名为 xigouhanshu。实例的功能是通过调用析构函数输出指定的文本。实例文件 Program.cs 的主要代码如下：

```
namespace xigouhanshu
{
    class mm
    {
        public mm()
        {
            Console.WriteLine("mm 的构造函数");

        }
        ~mm()
        {
            Console.WriteLine("mm 的析构函数");
        }
    }
    class B
    {
        public B()
        {
            Console.WriteLine("B 的构造函数");
```

```
            }
            ~B()
            {
                Console.WriteLine("B 的析构函数");
            }
            public void nn()
            {
                Console.WriteLine("进入 B.nn()");
                mm bb = new mm();
                Console.WriteLine("退出 B.nn()");
            }
            static void Main()
            {
                Console.WriteLine("进入 Main（）");
                B cc = new B();
                cc.nn();
                Console.WriteLine("按 Enter 键退出程序...");
                Console.Read();
                Console.WriteLine("退出 Main（）");

            }
        }
    }
```

上述实例代码的设计流程如下。

1）分别定义类 mm 和 B。

2）在类 mm 内分别定义静态构造函数 mm()和析构函数 mm()。

3）在类 B 内定义静态构造函数 B()和析构函数～B()。

4）定义函数 nn()，输出指定的文本。

5）定义类 B 的对象实例 cc，输出对应的结果。

执行后将首先输出显示默认的值，具体如图 8-6 所示。

```
进入Main ()
B的构造函数
进入B.nn()
mm的构造函数
退出B.nn()
按Enter键退出程序...
```

图 8-6　实例执行结果

当屏幕中输出“按 Enter 键退出程序...”时，B.nn()已经执行完毕，类 A 的对象 bb 已经无效。但是没有看见 bb 的析构函数被执行，既没有打印出“执行 A 的析构函数”。只有在退出程序之后，bb 的析构函数才被执行，即先打印“退出 Main()”，后打印“执行 A 的析构函数”。这说明，类 A 的对象 bb 在退出 B.nn()后就无效了，它的析构函数就随时可能被调用，并不能确定它的调用时机。

多学一招

为了便于读者掌握，在此将 C#构造函数和析构函数进行简要总结。

- 析构函数和构造函数正好相反，构造函数和类同名，没有返回值。
- 析构函数在类名前加~，也没有返回值。

- 构造函数是在对象创建时执行。
- 析构函数是在程序结束时执行。
- 一般情况下析构函数里面写的都是一些资源回收之类的东西。
- C#析构函数的调用机制和 C++不同，并不能保证每次都会调用，所以最好不要利用 C#的析构函数来回收资源。

8.8　使用只读字段

在 C#中，使用关键字 readonly 来修饰只读字段。通过使用只读字段，能够设置程序内指定字段的数据不可更改。只读字段只能在如下两个位置中被赋值或传递到方法中被改变。

1）在声明字段时赋值。

2）在构造函数内可被多次赋值，在类的构造函数内被赋值或传递到方法中被改变。

只读字段和常量非常类似。如果一个值在整个程序中保持不变，并且在编写程序时就已经知道这个值，那么就应该使用常量。如果这个值在编写程序时不知道，而是程序运行时才能得到，而且一旦得到这个值，其值就不会再改变，那么就应该使用只读变量。比如，在上面的 mm 类中，在编写程序时，假如 NUM 是已知的，并且不会改变，因此会使用常量；而 nn 在编写程序时是不知道的，但是一旦程序开始运行了，这个值就不会再变化，因此使用只读字段。



常量与只读字段有如下三点区别。

1）常量只能在声明时赋值，常量的值在编译时就已经确定，在程序中根本不能改变。

2）只读字段可以在声明时或者在构造函数内赋值，只读字段的值不能在编译时确定，而是在运行时确定的。

3）常量的类型只能是下列类型之一：sbyte、byte、short、ushort、int、uint、long、ulong、char、float、double、decimal、bool、string 或者枚举类型；而只读字段可以是任何类型。

8.9　使用 this 关键字

C#中的 this 关键字有特殊的功效，其中最基本的功能有如下两点。

1）this 访问。

2）在声明构造函数时指定最先执行的构造函数。

接下来开始详细讲解上述功能的基本知识。

8.9.1　进行 this 访问

在类内的实例构造函数和实例函数成员中，关键字 this 表示当前的类实例或对象的引用。但是不能将关键字 this 用在静态构造函数和静态函数成员中，也不能在其他地方使用。

当在实例构造函数或方法内使用了与字段名相同的变量名或参数名时，可以使用 this 关键字来区别字段、变量或者参数。看下面的一段代码：

```
public class Dog{
   public string name;
   public int age;
   public mm()
   {
   }
   public mm(string name)                  // 这个 name 是指传入的参数 name
   {
      this.name = name;                    // 这个 name 表示的是字段 name
   }
   public mm(string name, int age)         // 这个 name 是指传入的参数 name
   {                                       // 这个 age 是指传入的参数 age
      this.name = name;                    // 这个 name 表示字段 name
      this.age = age;                      // 这个 age 表示字段 age
   }
}
```

从本质上讲，this 被定义为一个常量。所以说虽然在类的实例构造函数和实例函数成员中，this 可以用于引用该函数成员调用所涉及的实例，但是不能对 this 本身赋值或者改变 this 的值。例如，使用 this++和--this 等格式的操作是非法的。

8.9.2　使用 this 关键字

在 C#构造函数中，使用 this 关键字的格式如下：

```
访问修饰符 类名(形参数) : this(实际参数)
     {
     处理语句
     }
```

在上述格式中，this 的功能是指定该类本身所声明的、形式参数表与实际参数表最匹配的另一个实例构造函数，这个构造函数会在执行正在声明的构造函数之前执行，即优先执行。

实例 26：按照顺序输出构造函数的执行结果

下面通过一个具体的实例来说 this 在构造函数内的使用流程。本实例保存在“光盘：\daima\8”文件夹内，项目名为 thisguanjian。实例的功能是按照顺序输出构造函数的执行结果。实例文件的主要代码如下：

```
namespace thisguanjian {
    class PuHuoClass
    {
        public PuHuoClass(int a)
        {
            Console.WriteLine("PuHuoClass.PuHuoClass(int a)");
        }
        public PuHuoClass(string s, int n)
            : this(0)
```

```
            {
                Console.WriteLine("PuHuoClass.PuHuoClass(string a, int b)");
                Console.ReadKey();
            }
        }
        class nn
        {
            static void Main()
            {
                PuHuoClass a = new PuHuoClass("PuHuoClass的Class", 1);
            }
        }
    }
```

上述实例代码的设计流程如下。

1）分别定义类 PuHuoClass 和 nn。

2）声明一个关键字的构造函数 PuHuoClass (int a)。

3）声明两个关键字的构造函数 PuHuoClass (string s, int n)。

4）定义实例对象 a。

执行后将输出显示构造函数的执行结果，如图 8-7 所示。

```
PuHuoClass.PuHuoClass(int a)
PuHuoClass.PuHuoClass(string a, int b)
```

图 8-7　实例执行结果


多学一招

在 C#中使用 this 关键字时可以遵循如下两点技巧:

1）当参数和类变量同名时必须用 this。例如:

```
class Demo {
private  int  i;
Demo(){}
Demo(int i)
{
this.i = i;  //必须用this
}
}
```

2）使用时避免自己给自己复制。例如:

```
Class Demo {
public void XXX(Demo a) {
if(this == a)
...
else
```

```
    ...
    }
    }
```

8.10 C#属性

属性是对现实世界中实体特征的抽象，提供了对类或对象性质的访问。类属性描述了一种状态信息，在类的某个实例中，属性的值表示该对象的状态值。属性和字段是密切相关的，属性是字段的扩展，两者都是具有关联类型的命名成员，而且对两者进行访问的语法格式也是相同的。

但是，属性不表示存储位置，这和字段是不同的。在使用属性时，可以通过访问器指定当它们的值被读取或写入时需执行的语句。属性提供了一种机制，可以把读取和写入对象的某些性质与一些操作关联起来，甚至还可以对此类性质进行计算。属性的声明格式如下：

```
修饰符 类型 属性名
{
    get
        {
…
      }
    set
        {
…
        }
}
```

其中，修饰符是可选的，最为常用的就是访问修饰符，也可以使用 static 修饰符。

当属性声明包含 static 修饰符时，则被称为静态属性。当不存在 static 修饰符时，则被称为实例属性。具体说明如下。

1）静态属性不与特定实例相关联，因此在静态属性的访问器内引用 this 会导致编译错误。

2）实例属性与类的一个给定实例相关联，并且可以在属性的访问器内通过 this 来访问该实例。

属性的类型可以是任何的预定义或者自定义类型。属性名是一种标识符，命名规则与字段相同。

C#中的属性通过 get 和 set 访问器来对属性的值进行读写。get 和 set 访问器分别用关键字 get 和 set，以及位于一对大括号内的代码块构成。代码块内的代码用于分别指定调用相应访问器时需执行的语句块。

实例 27：输出类成员的操作结果

下面通过一个具体的实例来说使用属性的流程。本实例保存在“光盘：\daima\8”文件夹内，项目名为 shiyongshuxing。实例的功能是输出类成员的操作结果，实现文件的主要代

码如下：

```
namespace shiyongshuxing {
    public class mm
    {
        protected int kuan;
        protected int gao;
        public int Width
        {
            get
            {
                return kuan;
            }
            set
            {
                if (value > 0)
                    kuan = value;
                else
                    Console.WriteLine("不能为负数。");
            }
        }
        public int Height
        {
            get
            {
                return gao;
            }
            set
            {
                if (value > 0)
                    gao = value;
                else
                    Console.WriteLine("不能为负数。");
            }
        }
        public int Mianji
        {
            get
            {
                return kuan * gao;
            }
        }
        public mm()
        {
        }
        public mm(int cx, int cy)
        {
```

```
                Width = cx;
                Height = cy;
            }
            public static void Main()
            {
                mm rect = new mm();
                rect.Width = 2;
                rect.Height = 4;
                Console.WriteLine("宽 = {0}, 高 = {1}, 面积 = {2}", rect.Width,
    rect.Height, rect.Mianji);
                rect.Width = -2;
                rect.Height = -4;
                Console.WriteLine("宽 = {0}, 高 = {1}, 面积 = {2}", rect.Width,
    rect.Height, rect.Mianji);
                mm rect1 = new mm(-4, -6);
                Console.WriteLine("宽 = {0}, 高 = {1}, 面积 = {2}", rect1.Width,
    rect1.Height, rect1.Mianji);
                Console.ReadKey();
            }
        }
    }
```

上述实例代码的设计流程如下。

1）定义类 mm。

2）分别定义两个读写属性 kuan 和 gao，并设置返回参数。

3）定义一个 public 属性 Mianji，赋值为参数 kuan 和 gao 的积。

4）分别定义两个构造函数。

5）分别创建对象实例 rect 和 rect1，并对参数进行赋值。

6）根据所赋的对应参数值，输出各参数值。

上述实例代码执行后，将各参数值输出显示，如图 8-8 所示。

```
宽 = 2, 高 = 4, 面积 = 8
宽的值不能为负数。
高的值不能为负数。
宽 = 2, 高 = 4, 面积 = 8
宽的值不能为负数。
高的值不能为负数。
宽 = 0, 高 = 0, 面积 = 0
```

图 8-8　实例执行结果

在上述实例中，定义了一个 mm 类，其高和宽都被定义为 protected 字段，这样既保证了数据隐藏，又能满足继承它的子类的需要。为了获取并设置高和宽的值，分别定义了两个读写属性 Height 和 Width。mm 还包含一个只读属性 mianji 来读取面积值。面积是通过高和宽计算得到的，但是不能被赋值，因此使用只读属性，以避免错误赋值。

在 Height 和 Width 的 set 访问器中检查了所赋的值是否合理，如果高和宽的值不合理，

则会输出指定的提示。由此可以看出，属性的引用方法与字段完全一样，因此使用非常方便。但是应该牢记的是，属性本质上是方法，而不是数据成员。


多学一招

get 和 set 访问器是可选的，在使用时必须注意如下四点。

1）get 访问器相当于一个具有属性类型返回值的无参数方法。除了作为赋值的目标，当在表达式中引用属性时，将调用该属性的 get 访问器以计算该属性的值。get 访问器体必须用 return 语句来返回，并且所有的 return 语句都必须返回一个可隐式转换为属性类型的表达式。

2）set 访问器相当于一个具有单个属性类型值参数和 void 返回类型的方法。set 访问器的隐式参数始终命名为 value。当一个属性作为赋值的目标，或者作为运算操作数值被引用时，就会调用 set 访问器，所传递的参数将提供新值。

3）不允许 set 访问器体中的 return 语句指定表达式。由于 set 访问器隐式具有名为 value 的参数，因此在 set 访问器中不能自定义使用名称为 value 的局部变量或常量。

4）由于属性的 set 访问器中可以包含大量的语句，因此可以对赋予的值进行检查，如果值不安全或者不符合要求，就可以进行提示。这样就可以避免因为给类的数据成员设置了错误的值而导致的错误。

8.11　索引器

索引器是属性的一种特例，在 C#中可以通过索引来操作数组内的元素。C#索引器能够用引用数组元素的方式来引用对象。

C#中的索引器通常被称为带参数的属性，其声明方式与属性的声明方式类似。其具体格式如下：

```
修饰符 类型 this[参数]
{
   get
       {
       }
   set
       {
       }
}
```

此处的属性声明比较特殊，具体说明如下。

1）索引器的名称必须是关键字 this。

2）this 后面一定要跟一对方括号“[ ]”，在方括号之间指定索引的参数表，其中至少必须有一个参数。

3）索引器不能被定义为静态成员，而只能是实例成员。

4）其他的定义都与属性完全一样，也是通过 get 访问器来取值，通过 set 访问器来赋值。

5）索引器使用方式不同于属性的使用方式，需要使用元素访问运算符[]，并在其中指定参数来进行引用。

实例 28：输出指定数组内的数据

下面通过一个具体的实例来说明使用索引器的流程。本实例保存在“光盘：\daima\8”文件夹内，项目名为 shiyongsuoyin。实例的功能是输出指定数组内的数据。实例文件的主要代码如下所示：

```
namespace shiyongsuoyin {
   class mm
   {
      public const int MaxNum = 8;
      private string[] member;
      public string this[int idx]
      {
         get
         {
            return member[idx];
         }
         set
         {
            member[idx] = value;
         }
      }
      public mm()
      {
         member = new string[MaxNum];
      }
      public static void Main()
      {
         mm group = new mm();
         group[0] = "ＡＡ";
         group[1] = "ＢＢ";
         group[2] = "ＣＣ";
         group[3] = "ＤＤ";
         group[4] = "ＥＥ";
         group[5] = "ＦＦ";
         group[6] = "ＧＧ";
         group[7] = "ＨＨ";
         for (int i = 0; i < mm.MaxNum; ++i)
            Console.WriteLine(group[i]);
            Console.ReadKey();
      }
   }
}
```

上述实例代码的设计流程如下。

1）定义类 mm。

2）定义变量 MaxNum 和数组 member，分别用于设置数组成员个数和保存成员值。
3）使用 this 创建一个索引器，设置取值为索引序号，返回值为索引对应的成员名。
4）创建对象实例 mm，并给各索引成员赋值。
5）通过 for 语句逐一读取数组内的值。
6）按照索引顺序输出数组内的成员。
执行后将数组内的保存值输出显示，如图 8-9 所示。

```
file:///E:/机械工业/c#/daima/8/shiyongsuoyin/bi
AA
BB
CC
DD
EE
FF
GG
HH
```

图 8-9　实例执行结果

8.12　命名空间

在本书前面的内容中，已经多次用到过命名空间的知识。命名空间是 C#程序的重要组成部分。C#语言是通过命名空间组织起来的，命名空间既可以作为程序的内部的组织系统，也用作为外部的公开的组织系统。在本节的内容中，将对 C#命名空间的基本知识进行详细介绍。

8.12.1　编译单元

C#的编译单元定义了源文件的总体结构。一个 C#程序是由一个或多个编译单元组成的，每个编译单元都用独立的源文件来保存。编译 C#程序时，所有这些编译单元一起进行处理。因此，这些编译单元间可以互相依赖，甚至以循环方式互相依赖。

如果一个程序包含多个源文件，编译程序时则需要将所有的源文件放在一起编译，具体的命令格式如下：

```
csc <source 文件名>  <source 文件名> ... <source 文件名>
```

使用如下的命令可以显示 csc 的所有参数的意义：

```
csc /?
```

使用 Visual Studio 可以迅速地创建一个项目，并能将所有的源文件都放在项目中。
为了最大限度地避免类名冲突，C#通常使用命名空间来组织类。具体使用原则如下。
1）在同一个命名空间内，类名不能重名。
2）在不同的命名空间中，可以使用相同的类名。
一个编译单元可以包括任意命名空间定义，也可能没有。命名空间是可以嵌套的，即在

一个命名空间内部还可以定义其他的命名空间。

声明命名空间可以作为顶级声明出现在编译单元中，也可以作为成员声明出现在另一个命名空间声明内。当命名空间声明作为顶级声明出现在编译单元中时，该命名空间成为全局命名空间的一个成员。当某个命名空间声明出现在另一个命名空间声明内时，该内部命名空间就成为包含着它的外部命名空间的一个成员。无论是哪种情况，一个命名空间的名称在它所属的命名空间内必须是唯一的。

在 C#中，使用关键字 namespace 来声明命名空间，具体格式如下：

```
namespace 命名空间名
{
    using 指令
    代码块
}
```

命名空间的隐式关键字是 public，而且在命名空间的声明中不能包含任何访问修饰符。

在代码块内，可选用 0 个或者多个 using 指令来导入其他命名空间和类型的名称，这样就可以直接地而不是通过限定名来引用它们。

命名空间名称可以是单个标识符或者是由“.”标记分隔的标识符序列。当使用“.”标记时，允许一个程序直接定义一个嵌套命名空间，而不必按词法嵌套若干个命名空间声明。例如，下面的两段代码的功能是等效的：

```
namespace mm.nn
{
  class A {}
  class B {}
}
//////////////////////////////////////////////////////////////
namespace mm
{
  namespace nn
  {
    class A {}
    class B {}
  }
}
```

8.12.2　完全限定名标识

每个命名空间和类型都具有一个完全限定名，它可以通过该名称在所有其他命名空间或类型中唯一标识该命名空间或类型。假设存在一个命名空间 mm 或类型 mm，则完全限定名应该遵循如下两条规则。

1）如果 mm 是全局命名空间的成员，则它的完全限定名为 mm。

2）如果 mm 不是全局命名空间的成员，它的完全限定名为 A.mm，其中 A 声明了 mm 的命名空间或类型的完全限定名。

因为命名空间或类型的每个成员都必须具有唯一的名称，所以如果将这些成员名称置于命名空间或类型的完全限定名之后，这样构成的成员完全限定名肯定符合唯一性。

在下面的代码中，演示了多命名空间和类型声明及其关联的完全限定名。

```
class mm {}
namespace aa
{
  class nn
  {
    class C {}
  }
  namespace zz
  {
    class D {}
  }
}
namespace aa.nn
{
  class E {}
}
```

8.12.3　using 指令

通过使用 using 指令，简化了在其他命名空间中定义的命名空间和类型的使用步骤。using 指令能够影响命名空间或者类型名称的解析过程，但是不会声明为任何类型。为此，using 指令不会在使用它们的编译单元或命名空间中定义新成员。

C#中使用 using 指令的具体语法格式如下：

```
using 【命名空间名】;
```

命名空间名称可以是单个标识符或者是由“.”标记分隔的标识符序列。using 指令将【命名空间名】所标识的命名空间内的类型成员导入当前编译单元中，从而可以直接使用每个被导入的类型的标识符，而不必加上它们的完全限定名。

C#内 using 指令的主要特点如下。

1）在包含 using 指令的编译单元或命名空间体中的成员声明内，可以直接引用包含在给定命名空间中的那些类型。

2）using 指令能够导入包含在给定的命名空间中的类型，但是不能导入该命名空间所嵌套的命名空间。

3）using 指令还可以使用指定别名的形式，具体格式如下：

```
using 【别名】 = 【命名空间或类型名】;
```

4）using 指令中的【别名】标识符在如下条件时必须是唯一的。

- ❑ 直接包含该 using 指令的编译单元。

❑ 命名空间的声明空间内时。

看下面的一段代码：

```
namespace N3
{
  class A {}
}
namespace N3
{
  using A = N1.N2.A;
}
```

在上述代码中，class A 是可行的，因为 A 已经定义在命名空间 N3 中；但是“using A = N1.N2.A”是非法的，因为 A 已经存在于命名空间 N3 中。

5）using 指令指定的别名是不可传递的，它仅影响使用它的编译单元或命名空间体，而不会影响具有相同限定名的命名空间。

6）using 指令的顺序并不重要。using 指令可以为任何命名空间或类型创建别名，包括它所处的命名空间，以及嵌套在该命名空间中的其他任何命名空间或类型。

7）对一个命名空间或类型进行访问时，无论用它的别名，还是用它所声明的名称，最终结果是完全相同的。

8）C#内的 using 指令导入的名称会被如下元素隐藏。

❑ 包含该指令的编译单元。

❑ 命名空间体中具有相同名称的成员。

8.13 C#内部类

可以嵌套定义 C#的类，即可以在某个类的内继续定义其他的类。在类内声明的类被称为内部类或嵌套类，在编译单元或命名空间内声明的类被称为顶级类或非嵌套类型。看下面的代码，在 mm 类中定义了一个 private 类型的内部类 nn。

```
public class mm
{
  private class nn
  {
    public object Data;
    public Node Next;
    public Node(object data, Node next)
    {
      this.Data = data;                      // this 是 Node 类的对象
      this.Next = next;
    }
  }
  private Node first = null;
  private Node last = null;
```

```
    public void AddToFront(object o) {...}
  }
```

内部类和包含它的那个类并不具有特殊的关系。在内部类内，this 不能用于引用包含它的那个类的实例成员，而只能引用内部类自己的成员。例如在上述代码中的内部类 nn 中的 this，只能引用 nn 的对象，而不能代表 mm 类的对象。

C#内部类的主要特点如下。

1）如果类 A 是类 B 的内部类，当需要在内部类 A 的内部访问类 B 的实例成员时，可以在类 B 中将代表类 B 的实例的 this 作为一个参数传递给内部类 A 的构造函数，这样就可以实现在类 A 的内部对类 B 的访问。

2）非嵌套类可以具有 public 或 internal 访问修饰符，默认的访问修饰符是 internal。而内部类可以是五种访问修饰符中的任何一种，而且与其他类成员一样，默认的已访问修饰符是 private。

3）内部类的可访问域受包含它的类的访问修饰符和它自身的访问修饰符的限制。内部类的可访问域至少为包含它的类体。内部类可访问域是声明它的类的可访问域的子集。

4）内部类的成员的可访问域受包含内部类的类访问修饰符、内部类的访问修饰符和它自身的访问修饰符的限制。内部类成员的可访问域是内部类的可访问域的子集。

8.14 C#分部类

程序员需要遵循的一个好的编程习惯是，在单个文件中维护某个类的所有源代码。但是有时一个类会变得非常大和复杂，此时单个文件维护就成为一种不切实际的限制。此外，程序员经常使用源代码生成器来生成应用程序的初始结构，然后修改得到代码。遗憾的是，当将来某个时候再次用代码生成器来生成源代码时，已有的修改会被源代码生成器改写或删除。为解决上述问题，分部类便被提上了日程。

分部类可以将一个类划分为多个部分，存储在不同的源文件中，以便于开发和维护。此外，分部类允许将计算机生成的类型部分和用户编写的类型部分互相分开，以便更容易地扩充工具生成的代码。

在 C# 4.0 中，新增了类修饰符 partial，用来实现通过多个部分来定义一个类。partial 修饰符必须直接放在 class 关键字的前面。C#分部类主要具有如下特点。

1）分部类声明的每个部分都必须包含 partial 修饰符，并且其声明必须与其他部分位于同一命名空间。partial 修饰符说明在其他位置可能还有同一个类型声明的其他部分，但是这些其他部分并非必须存在；如果只有一个类型声明，包含 partial 修饰符也是有效的。

2）当分部类声明指定了可访问性时，它必须与所有其他部分所指定的可访问性一致。

3）分部类的所有部分必须一起编译，以使这些部分可在编译时被合并。但是，分部类不允许用于扩展已经编译的类型。

4）partial 修饰符可以用于在多个部分中声明嵌套类型。通常，其包含类型也使用 partial 声明，并且嵌套类型的每个部分均可在该包含类的不同部分中声明。

5）在多个部分中声明的类的成员是每个部分中声明的成员的并集。所有部分的类声明主体都表示同一个类，并且每个成员的范围都扩展到所有部分的主体。任何成员的可

访问性域都包含类的所有部分；在一个部分中声明的 private 成员可从其他部分随意访问。在类的多个部分中声明同一个成员将引起编译时的错误，除非该成员是带有 partial 修饰符的类型。

8.15 疑难问题解析

本章详细介绍了 C#中类的基本知识。本节将对本章中比较难以理解的问题进行讲解。

读者疑问：请简要说明 C#类中的值类型？

解答：各种值类型总是含有相应该类型的一个值。C#必须初始化变量才能使用它们进行计算-变量没有初始化不会出问题，因为当企图使用它们时，编译器会提示。每当把一个值赋给一个值类型时，该值实际上被复制了。相比，对于引用类型，仅是引用被复制了，而实际的值仍然保留在相同的内存位置，但现在有两个对象指向了它（引用它）。C#的值类型可以归类如下。

- 简单类型（Simple Types ）。
- 结构类型（Struct Types）。
- 枚举类型（Enumeration Types）。

读者疑问：请简要介绍 C#数据类型之间的转换的操作技巧。

解答：首先来看 int 和 String 之间的转换 int->String，String->int 就有很多选择，例如 ToInt16()，ToInt32，ToInt64 等的类似的函数，他们的用法都是一样的。如何把 String 根据指定的规则转换成数组类型，如何在一个 String 中将指定的 String 转换成另外的 String，例如将 “'” 转换成 “''”。在这里需要引入一个新的 Class StringBuilder，来看如下的函数:

```
<script language="C#" runat=server>
function replaceStr1ToStr2(String str1,String str2,String str3){
//str1 是包含有要替换的字符串的 原字符串
//str2 是要替换的字符串
//str3 是要将 str2 替换成的 str3
StringBuilder d=new StringBuilder(str1);
d=d.Replace(str2,str3);
return d.ToString();
}
</script>
```

然后，就可以通过 replaceStr1ToStr2(str,"'","''")处理 SQL 语句的时候处理 “'” 符号了。另外还可以通过 replaceStr1ToStr2(str,"\n","
") 来处理输出数据的时候的换行符。



职场点拨——谈职业精神

说起职业程序员，可能大家脑子里马上反应出的是西服领带公文包的精干白领形象，其

实作为程序员来说，有更多内在的职业精神。读者应该怎样提高自己的职业精神呢？这首先需要要掌握职业精神的基本要素，职业精神的基本要素如下。

（1）职业理想

职业理想是人们实现个人生活理想、道德理想和社会理想的手段，并受社会理想的制约。职业理想是人们对职业活动和职业成就的超前反映，与人的价值观、职业期待、职业目标密切相关，与世界观、人生观密切相关。一般说来，从业者对职业的要求可以概括为如下三个方面:

- 维持生活。
- 完善自我。
- 服务社会。

在择业时应该把服务社会放在首位。因为，只有从社会的整体利益出发，分别从事社会所需要的各种职业，社会才能顺利地前进和发展。只有在这个基础上，广大社会成员包括从业者自身，才能过上幸福的生活。

（2）职业态度

职业态度是指个人职业选择的态度，包括选择方法、工作取向、独立决策能力与选择过程的观念，简而言之，职业态度就是指个人对职业选择所持的观念和态度。职业态度很重要，树立正确的职业态度是从业者做好本职工作的前提。作为一名程序员，要意识到一个从业者积极性的高低和完成职业的好坏，在很大程度上取决于他的职业价值观念。

（3）职业责任

职业责任是指人们在一定职业活动中所承担的特定的职责，它包括人们应该做的工作和应该承担的义务。主要包括两个方面，分别是职业团体责任和从业者个体责任。作为一名程序员，虽然地位仅仅是一名职员，但是也应该自觉明确和认定自己的职业责任，树立职业责任。企业对员工职业责任教育主要通过以下途径：以质量观念促进责任意识；完善各项岗位规章制度。

（4）职业技能

根据我国的具体情况，可以将职业技能定义为，它是按照国家规定的职业标准，通过政府授权的考核鉴定机构，对劳动者的专业知识和技能水平进行客观公正、科学规范地评价与认证的活动。职业技能，即指就业所需的技术和能力。

作为一名程序员，要意识到随着当前竞争压力的增大，对职业技能的要求也越来越高。不但需要高技术专家人才，而且迫切需要千百万受过良好职业技术教育的初、中级技术人员、管理人员，甚至是技工和其他具有一定科学文化知识和技能的熟练从业者。如果没有一线劳动者大军，再先进的科学技术和先进的设备也会成为空谈。良好的职业技能具有深刻的职业精神价值。

（5）职业纪律

职业纪律是劳动者在从业过程中必须遵守的从业规则和程序，它是保证劳动者执行职务、履行职责、完成自己承担的工作任务的行为规则。职业纪律可以保障从业者的自由，所以职业纪律虽然有强制性的一面，但更有为从业者的内心信念所支持、自觉遵守的一面，而且这是主要的一面，从而具有丰富的精神内涵。作为一名程序员，人们要服从职业的要求，这种职业要求是法规性和道德性的统一。

（6）职业良心

职业良心是指有着特殊职业的从业人员领悟了社会对自己的要求，因而具有为社会尽具体义务的明确意识，或简单地说，就是从业人员对职业责任的自觉意识。职业良心能依据履行责任的要求，对行为的动机进行自我检查，对行为活动进行自我监督。作为一名程序员，要按照用户和公司的要求，制作出一个合法有用的项目程序，而不能触及公司和客户的利益。

（7）职业信誉

职业信誉是职业责任和职业良心的价值尺度，包括对职业行为的社会价值所做出的客观评价和正确的认识。职业信誉能使一个人自觉地按照客观要求的尺度去履行义务，宁愿做出自我牺牲也不愿违背职业良心，做出可耻、毁誉和损害职业精神的事情。作为一名程序员，要有自己的信誉，如果已经答应客户某些服务和功能，就一定要实现。良好的信誉是个人成功的关键，也是企业发展的关键。

（8）职业作风

职业作风，是指从业者在其职业实践和职业生活中所表现的一贯态度。程序员要养成良好的职业作风，为公司、为客户奉献出自己的作品，为客户行方便，为公司创造出利润和价值。

第 9 章　集合和数组应用

经过前面内容的学习，了解了 C#语言的基本知识和面向对象编程技术的思想。从本章的内容开始，将继续向读者讲解 C#体系中的基本知识，本章将介绍集合和数组的高级应用知识。集合可以对数组的功能进行扩展，并维护扩展的对象组。例如，控制对其包含对象的访问、搜索和排序等。在数组中应用集合知识，实现更为复杂的功能。通过本章能学到如下知识。

- C#集合和集合类。
- 数组的高级应用。
- 使用 foreach 循环语句。
- 迭代器。
- 职场点拨——老板喜欢什么样的员工。

2007 年 XX 月 X 日，天气阴。

今天的事情让我很为难，项目经理让我晚上加班，估计得忙到半夜，但是我很早就答应女友陪她去车站接她父母。真希望自己会七十二变，能够应付一切问题。



一问一答

小菜：我真的很为难啊，恨不能把自己掰成两半，一半去接女朋友父母，一半晚上加班！

Wisdom：呵呵，生活中我们总会遇到很多烦心事，职场中也不例外。此时需要你冷静下来想出一个好的办法。如果只是一味着急，是不能解决问题的。在本章最后的“老板喜欢什么样的员工”希望会帮助你！

小菜：嗯，言归正传，本章的数组有什么用？

Wisdom：集合和数组的意义重大，你在职场中会遇到很多问题，这需要你耗费额外的精力去解决。而 C#程序中，也需要面对各种不同的数据类型。有些数据通过变量和常量是不能解决的，为此 C#推出了集合和数组，这样就不用额外编程来实现特殊数据类型了。

9.1　C#集合

集合是和数组一起使用的，在本书第 3 章中已经介绍了数组的基本知识。在日常应用中，直接使用数组会有很大的限制，例如一旦定义并创建数组后，数组的内容就固定不变

了。要想在里面添加或删除数据，必须创建一个新的数组。而通过集合，可以把相互联系的数据组合到一个集合内，这样就能够有效地处理这些密切相关的数据。通过集合能够使用相同的代码来处理一个集合内的所有元素，而不需要编写不同的代码来处理单个独立的对象。

集合是组合在一起的数据组，在 System.Collections 命名空间内提供了一系列的接口，所有的集合类都实现了其中的一些接口。通过接口的实现，可以设计出自定义的集合类。

9.1.1　C#集合概述

在 C#中，数组的实现是 System.Array 类的实例，它们只是集合类的一种。集合类能够处理对象列表，它的功能要比数组强大。因为集合类通过实现 System.Collections 命名空间中的接口实现处理功能，所以其中集合的语法都符合标准化。

集合使用接口来实现功能，这个接口不仅没有限制使用的基本集合类，而且还可以创建一个自定义的集合类。这样从集合内提取数据时，就不需要把它们特意转换为专用的类型了。

在 System.Collections 命名空间中，可以通过如下接口提供基本集合功能。

- IEnumerable，用于迭代集合项。
- ICollection，能够获取集合项的个数，并把项复制到一个简单的数组类型中。
- IList，提供了集合项列表，并访问这些项和一些与项列表相关的功能。
- Idictionary，和 IList 类似，但是它能够通过键码值访问项列表。

其中，System.Collections. ICollection 是所有集合的基接口，具体声明格式如下：

```
public interface ICollection : IEnumerable{
}
```

因为 System.Collections. ICollection 继承了 System.Collections. IEnumerable，并没有添加任何的成员。所以可以完全等价于 System.Collections. IEnumerable 接口，即所有的集合类都必须实现 System. Collections. IEnumerable 接口。当类实现了这个接口后，就可以依次列举集合类内所包含的数据元素。IEnumerable 接口的定义格式如下：

```
public interface IEnumerable{
   IEnumerator GetEnumerator();
}
```

IEnumerable 接口只包含一个方法 GetEnumerator()，它返回一个能够访问集合数组的列举器对象。列举器必须实现 System.Collections.IEnumerator 接口，IEnumerator 的定义格式如下：

```
public interface IEnumerator{
    Boolean MoveNext();
    Object Current
      {
        get;
      }
    void Reset();
}
```

1．列举器

列举器能够读取集合中的数据，即循环访问集合的对象。但是列举器不能修改基础集合，实现 IEnumerator 接口的类必须实现 Reset()函数和 MoveNext()函数，以及 Current 属性。

当创建集合对象后，列举器应该首先定位在集合中的第一个元素前。Reset()函数将列举器返回到此位置，但此时调用 Current 属性会发生异常。所以在读取 Current 前必须调用函数 MoveNext，将列举器定位在集合的第一个元素。如果在调用 Reset()函数和 MoveNext()函数之前，Current 返回同一个对象，MoveNext()将 Current 设置为下一个元素。

虽然一个列举器只能与一个集合关联，但是一个集合可以有多个列举器。在 C#中，通常使用 foreach 语句操作列举器，同时隐藏了操作列举器的复杂性。

2．集合容量和集合计数

集合容量即集合所包含的元素数量，而集合计数是它实际包含的元素数目。在 System.Collections 命名空间中，集合在达到当前容量时会自动扩充容量。

3．集合下限

集合下限是它第一个元素的索引，在 System.Collections 命名空间中的集合下限都是 0。

9.1.2　使用集合

在 System.Collections 命名空间中，提供了接口 IList、ICollection 和 IEnumerable 来实现集合的功能。但是，它们只提供了一些需要执行的代码，例如 Clear()函数和 RemoveAt()函数。如果要实现特定的功能，还需要执行其他的指定代码。也可以使用 System.Collections.CollectionBase 类，这些类提供了许多集合类的实现方式。例如，可以使用如下代码定义对象 oop 的集合类。

```
public class Animals : CollectionBase
  {
    public void Add(Animal newAnimal)
    {
      List.Add(newAnimal);
    }
    public void Remove(Animal newAnimal)
    {
      List.Remove(newAnimal);
    }
    public Animals()
    {
    }
```

其中，Add()函数和 Remove()函数实现了强类型转化，List 接口用于访问项的标准 Add()函数。Add()函数只能处理 mm 类或它派生的类。

在 System.Collections.CollectionBase 的派生类中，可以使用 foerach 读取集合内的数据。例如：

```
static void Main(string[] args)
```

```
            {
              oop AA = new mm();
              AA.Add(new Cow("Jack"));
              AA.Add(new Chicken("Vera"));
              foreach (mm myAnimal in AA)
              {
                myAnimal.Feed();
              }
              Console.ReadKey();
            }
```

9.2　数组的高级应用

在本书第 3 章的内容中，已经简要介绍了数组的基本知识。在本节的内容中，将结合集合的基本知识，详细介绍 C#数组的高级应用知识。

9.2.1　数组的方法和属性

数组是经过索引后得到的相同类型对象的集合。因为 C#的数组都是从 System.Array 类中继承而来的，所以数组也继承了这个类的方法和属性。

1．数组属性和方法介绍

C#数组的常用属性如表 9-1 所示。

表 9-1　C#数组的常用属性

属　　性	描　　述
IsFixSize	获取一个值，设置 Array 是否具有固定大小
IsReadOnly	获取一个值，设置 Array 是否为只读
IsSynchronized	获取一个值，设置是否对 Array 同步访问
Length	获取一个 32 位整数，设置 Array 中维数的元素个数
Long Length	获取一个 64 位整数，设置 Array 中维数的元素个数
Rank	获取可同步访问 Array 的对象
SyncRoot	获取一个值，设置 Array 是否为只读

数组能够实现 IEnumerable 和 IEnumerator 的接口，数组的 GetEnumerator()方法能够返回数组的 IEnumerator 对象，得到的 IEnumerator 可以读取数组元素，但是不能修改数组元素。

C#数组的常用方法如表 9-2 所示。

表 9-2　C#数组的常用方法

方　　法	描　　述
BinarySearch()	使用二进制搜索方式在一维排序的 Array 中搜索
Clear()	设置 Array 元素为 0、false 或空引用
Clone()	创建 Array 的浅表副本

（续）

方　法	描　述
Copy()	将某 Array 内的元素复制到另一个 Array 中
CopyTo()	将当前一维 Array 的元素复制到另一维 Array 中
CreateInstance()	初始化 Array 的新实例
GetEnumerator()	返回 Array 的 IEnumerator
GetLength()	获取一个 32 位整数，设置 Array 指定维的元素数
GetLongLength()	获取一个 64 位整数，设置 Array 指定维的元素数
GetLowerBound()	获取 Array 中指定维度的下限
GetUpperBound()	获取 Array 中指定维度的上限
GetValue()	获取当前 Array 中指定元素的值
IndexOf()	返回一维获取 Array 或部分 Array 中某值第一个匹配项的索引
Initialize()	调用维类型的默认构造函数，初始化值类型获取 Array 的各元素
LastIndexOf()	返回一维获取 Array 或部分 Array 中某值最后一个匹配项的索引
Reverse()	反转一维 Array 部分 Array 中元素的顺序
SetValue()	将当前 Array 中指定元素设置为指定
Sort()	对一维 Array 对象中的元素进行排序

2．读取数组数据

实例 29：输出显示数组内指定的文本数据

下面将通过一个简单实例来说明。利用数组的属性和方法获取数组数据的具体流程。本实例保存在“光盘：\daima\9”文件夹内，项目名为 readshushu。实例的功能是输出显示数组内的指定文本数据。实例文件的主要代码如下：

```
namespace readshushu
{
    class Program
    {
        public static void Main()
        {
            // 创建和初始化一个数组
            String[] myArr = new String[10];
            myArr[0] = "ＡＡ";
            myArr[1] = "ＢＢ";
            myArr[2] = "ＣＣ";
            myArr[3] = "ＤＤ";
            // 显示数组
            int i = 1;
            System.Collections.IEnumerator myEnumerator = myArr.Get
Enumerator();
            Console.WriteLine("数组包含如下的值:");
            while ((myEnumerator.MoveNext()) && (myEnumerator.Current !
= null))
            {
```

```
                Console.WriteLine("第{0}： {1}", i++, myEnumerator.Current);

            }
            Console.ReadKey();
        }
    }
}
```

上述实例代码的设计流程如下。

1）定义 String 类型的数组 myArr。

2）设置 0～3 编号的数组值。

3）通过 GetEnumerator()方法逐一获取数组值。

4）通过 WriteLine()输出数组内的数据。

执行后将逐一显示数组内的数据，如图 9-1 所示。

```
file:///E:/机械工业/c#/daima/9/readsh
数组包含如下的值:
第1:  AA
第2:  BB
第3:  CC
第4:  DD
```

图 9-1　实例执行结果

3. 复制数组数据

实例 30：利用数组的属性和方法实现数组数据复制

下面将通过一个简单实例来说明利用数属性和方法实现数组数据复制的流程。来实例保存在“光盘：\daima\9”文件夹内，项目名为 copy。实例的功能是复制指定数组内的数据，并将复制结果输出。实例文件的主要代码如下所示：

```
namespace copy
{
    class Program
    {
        public static void Main()
        {
            int[,] array = new int[3, 2];
            Console.WriteLine("源数组：");
            for (int i = 0; i < array.GetLength(0); ++i)
            {
                for (int j = 0; j < array.GetLength(1); ++j)
                {
                    array[i, j] = i * j;
                    Console.Write("{0}  ", array[i, j]);
                }
                Console.WriteLine();
            }
            int[,] oppa = new int[3, 2];
            int[,] oppb = new int[2, 2];
            System.Array.Copy(array, oppa, array.Length);
            System.Array.Copy(array, 0, oppb, 0, oppb.Length);
            Console.WriteLine("复制数组 a：");
            for (int i = 0; i < oppa.GetLength(0); ++i)
            {
                for (int j = 0; j < oppa.GetLength(1); ++j)
```

```
                {
                    Console.Write("{0}  ", oppa[i, j]);
                }
                Console.WriteLine();
            }
            Console.WriteLine("复制数组 b: ");
            for (int i = 0; i < oppb.GetLength(0); ++i)
            {
                for (int j = 0; j < oppb.GetLength(1); ++j)
                {
                    Console.Write("{0}  ", oppb[i, j]);
                }
                Console.WriteLine();
            }
            Console.ReadKey();
        }
    }
}
```

上述实例代码的设计流程如下。

1）定义数组 array。

2）使用 for 语句逐一读取数组内的数据，并将数据输出。

3）分别定义数组 oppa 和 oppb。

4）将 array 内的数据复制到 oppa 中，并输出结果。

5）将 array 内的数据复制到 oppb 中，并输出结果。

执行后将显示指定数组内的数据，并输出复制后的结果，如图 9-2 所示。

```
file:///E:/机械工业/c#/daima/9/copy/bin/Debug/fuzhi.EXE
源数组:
0  0
0  1
0  2
复制数组a:
0  0
0  1
0  2
复制数组b:
0  0
0  1
```

图 9-2　实例执行结果

因为数组 mma 和 mmb 中允许的数据数不同，所以复制后最终的处理结果也不相同。

多学一招

除了使用 Array.Copy()方法之外，还可以使用 Clone()方法来创建浅副本。但 Clone()方法和 Copy()方法有一个重要区别，即 Clone()方法会创建一个新数组，而 Copy()方法只是传送了阶数相同、有足够元素空间的已有数组。如果需要包含引用类型的数组的深副本，就必须迭代数组，创建新对象。

4．数组数据搜索

通过数组方法 IndexOf()和 LastIndexOf()，可以在数组内搜索指定的数据。

实例 31：实现数组内数据搜索处理

下面将通过一个简单的实例来说明在数组内实现数据搜索处理的流程。本实例保存在“光盘：\daima\9”文件夹内，项目名为 search。实例的功能是输出显示数组内指定数据的索引。实例文件的主要代码如下：

```
namespace search {
    class Program
    {
        public static void Main()
        {
            int[] array = new int[10];
            Console.WriteLine("源数组：");
            for (int i = 0; i < array.Length; ++i)
            {
                array[i] = i;
                Console.Write("{0}  ", array[i]);
            }
            int n1 = System.Array.IndexOf(array, 5);
            Console.WriteLine("\n 第一个 5 的索引为{0}  ", n1);
            int n2 = System.Array.LastIndexOf(array,8);
            Console.WriteLine("最后一个 8 的索引为{0}  ", n2);
            Console.ReadKey();
        }
    }
}
```

上述实例代码的设计流程如下。

1）定义数组 array，设置有 20 个数据。

2）使用 for 语句逐一读取数组内的数据，并将数据输出。

3）通过 IndexOf()获取数组内第一个 4 的索引。

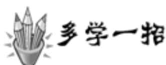
4）通过 LastIndexOf()获取数组内最后一个为 6 的索引。

5）输出显示获取的索引值。

执行后将显示指定数组内的数据，并将数组内指定数据的索引输出显示，如图 9-3 所示。

```
file:///E:/机械工业/c#/daima/9/search/bin/Debug/s
源数组：
0  1  2  3  4  5  6  7  8  9
第一个4的索引为5
最后一个6的索引为8
```

图 9-3　实例执行结果



利用索引进行查询的方法在实际项目中经常会被用到，比如在网站中编写上传图片时需要判断上传文件的扩展名，就需要用到 LastIndexOf（""），在上传文件的路径中，查找出索引位后，利用截取方法把文件路径中最后一个“”后的扩展名截取下来，再判断是否为合法的图片格式。

9.2.2　CreateInstance()方法

CreateInstance()方法能够初始化项目数据的新实例。在 C#开发应用中， CreateInstance()方法主要具有如下两个作用。

1．创建下限不为 0 的数组

在大多情况下，C#数组下限的默认值为 0。但是通过 System.Array 类提供的方法 CreateInstance()，可以创建指定的动态数组，也可以创建下限不为 0 的数组，从而重新声明了指定数组的大小。

实例 32：使用 CreateInstance()方法创建下限不为 0 数组

本实例将使用 CreateInstance()方法创建下限不为 0 数组。本实例保存在“光盘：\daima\9”文件夹内，项目名为 gaibian。本实例的功能是输出显示 CreateInstance()方法创建的下限不为 0 数组数据。实例文件的主要代码如下：

```
namespace gaibian
{
    class Program
    {
        public static void Main()
        {
            // 创建二维 string 类型数组
            int[] mm = new int[2] { 3, 5 }; // 第一维长度是 3， 第二维长度是 5
            int[] nn = new int[2] { 2, 3 }; // 第一维元素索引从 2 开始计数，第一维元素索引从 3 开始计数
            string[,] zz = (string[,])Array.CreateInstance(typeof(String), mm, nn); // 创建数组
            // 初始化二维 string 类型数组
            for (int i = zz.GetLowerBound(0); i <= zz.GetUpperBound(0); i++)
            {
                for (int j = zz.GetLowerBound(1); j <= zz.GetUpperBound(1); j++)
                {
                    int[] myIndicesArray = new int[2] { i, j };
                    zz.SetValue(Convert.ToString(i) + j, myIndicesArray);
                }
            }
            // 显示每一维的下限和上限
            Console.WriteLine("维数是\t 下限是\t 上限是");
```

```
            for (int i = 0; i < zz.Rank; i++)
                Console.WriteLine("{0}\t{1}\t{2}", i, zz.GetLowerBound(i),
zz.GetUpperBound(i));
            // 显示数组的元素
            Console.WriteLine("\n 数组元素是：");
            for (int i = zz.GetLowerBound(0); i <= zz.GetUpperBound(0); i++)
            {
                for (int j = zz.GetLowerBound(1); j <= zz.GetUpperBound
(1); j++)
                {
                    Console.Write("{0} ", zz[i, j]);
                }
                Console.WriteLine();
            }
            Console.ReadKey();
        }
    }
}
```

上述实例代码的设计流程如下。

1）分别定义二维数组 mm 和 nn。

2）通过 CreateInstance()方法创建一个新的数组 zz。

3）使用 for 语句逐一读取数组内的数据，将 zz 进行初始化处理。

4）输出显示数组的维数、下限和上限。

5）输出显示数组内的数据。

执行后将输出显示指定数组内的数据，并显示数组的维数、下限和上限，如图 9-4 所示。

```
维数是  下限是  上限是
0       2       4
1       3       7

数组元素是：
23 24 25 26 27
33 34 35 36 37
43 44 45 46 47
```

图 9-4　实例执行结果


多学一招

Array 类是一个抽象类，所以不能使用构造函数来创建数组。但除了可以使用 C#语法创建数组实例之外，还可以使用静态方法 CreateInstance()创建数组。如果事先不知道元素的类型，就可以使用该静态方法，因为类型可以作为 Type 对象传送给 CreateInstance()方法。

2．重定义数组

CreateInstance()方法的功能是，动态创建在编译时无法确定元素类型和维数的数组，即重新定义数组的大小。

实例 33：通过 CreateInstance()方法重新定义数组的大小

下面将通过一个简单的实例来说明，通过 CreateInstance()方法重新定义数组的大小的流程。本实例保存在“光盘：\daima\9”文件夹内，项目名为 redefine。本实例的功能是，输出显示指定数组内的数据，并使用 CreateInstance()输出此数组内指定长度的数据。实例文件的主要代码如下：

```
namespace redefine
```

```
{
    class Program
    {
        public static void Main()
        {
            int[] array = { 0, 1, 2, 3, 4, 5 , 6, 7, 8, 9};
            Console.WriteLine("数组长度为: {0}, 内容为: ", array.Length);
            shuchu((Array)array);
            Console.WriteLine("数组重新定义的长度为: {0}, 内容为: ", 12);
            array = (int[])ReDim(array, 12);
            shuchu((Array)array);
            Console.WriteLine("数组重新定义的长度为: {0}, 内容为: ", 4);
            array = (int[])ReDim(array, 4);
            shuchu((Array)array);
        }
        static Array ReDim(Array a, int newSize)
        {
            Array nn = Array.CreateInstance(a.GetType().GetElementType(),
newSize);
            int len = newSize;
            if (newSize > a.Length)
                len = a.Length;
            Array.Copy(a, 0, nn, 0, len);
            return nn;
        }
        static void shuchu(Array a)
        {
            for (int i = 0; i < a.Length; i++)
                Console.WriteLine("\ta[{0}]:{1}", i, a.GetValue(i));
                Console.WriteLine("");
                Console.ReadKey();
        }
    }
}
```

上述实例代码的设计流程如下。

1）定义一维数组 array，并在其中存储 10 个数字。

2）定义函数 shuchu()，输出指定长度的数组内数据。

3）定义函数 ReDim()，设置数组内的数据输出长度。

4）逐一输出此数组内指定长度的数据。

执行后将首先输出显示指定数组内的数据，如图 9-5 所示；当按〈Enter〉键后将显示长度为 12 的数组数据，如图 9-6 所示；当再次按〈Enter〉键后将显示长度为 4 的数组数据，如图 9-7 所示。

在上述实例中，如果设置的长度超出了数组内数据的长度，则将超出数据设置为 0 显示。

```
数组长度为：10，内容为：
        a[0]:0
        a[1]:1
        a[2]:2
        a[3]:3
        a[4]:4
        a[5]:5
        a[6]:6
        a[7]:7
        a[8]:8
        a[9]:9
```

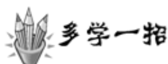
图 9-5　实例执行结果

```
数组重新定义的长度为：12，内容为：
        a[0]:0
        a[1]:1
        a[2]:2
        a[3]:3
        a[4]:4
        a[5]:5
        a[6]:6
        a[7]:7
        a[8]:8
        a[9]:9
        a[10]:0
        a[11]:0
```

图 9-6　实例执行结果

```
数组重新定义的长度为：4，内容为：
        a[0]:0
        a[1]:1
        a[2]:2
        a[3]:3
```

图 9-7　实例执行结果



CreateInstance()方法有许多重载版本，可以创建多维数组和不基于 0 的数组。例如在下面的代码中，创建了一个包含 2×3 个元素的二维数组。第一维基于 1，第二维基于 0:

```
int[] lengths = {2, 3};
int[] lowerBounds = {1, 10};
Array racers = Array.CreateInstance(typeof(Person), lengths, lowerBounds);
```

SetValue()方法用于设置数组的元素，其参数是每一维的索引：

```
racers.SetValue(new Person("Alain", "Prost"), 1, 10);
racers.SetValue(new Person("Emerson", "Fittipaldi"), 1, 11);
racers.SetValue(new Person("Ayrton", "Senna"), 1, 12);
racers.SetValue(new Person("Ralf", "Schumacher"), 2, 10);
racers.SetValue(new Person("Fernando", "Alonso"), 2, 11);
racers.SetValue(new Person("Jenson", "Button"), 2, 12);
```

9.2.3　数组转换

数组转换能够将某个数组转换为另外一种类型元素的数组，C#数组转换的基本原则如下：

1）值类型元素数组不能被转换为其他形式的数组。

2）如果数组的元素是引用类型，而且在被转换数组的元素和目标数组的元素类型之间，存在隐式转换或显式转换，那么数组可以被显式转换成维数相同的目标数组。

看下面的一段代码：

```
namespace zhuanhuan{
```

```
        public class mm
        {
           public int width;
           public int height;
           public mm()
            {
            }
           public mm(int w,int h)
           {
              width = w;
              height = h;
           }
        }
        class nn:mm
        {
           public nn(int x)
              {
              width=height=x;
              }
        }
        class Text
              {
            public static void Main()
              {
               nn[,] nns=new nn[3,2];
               mm[,] rcs1=(mm[,])nns;
               mm[] rcs2=(mm[])nns;
               object[,] aa=(object[,])nns;
               Type[,] types=(Type[,])nns;
               int[] ints=new int[8];
               object[] bb=(object[])nns;
              }
           }
     }
```

上述代码的设计流程如下。

1）分别设计三个类 mm、nn 和 Text，其中 mm 是 nn 的基类。

2）设置类 mm 和 nn 的元素成员类型为 int。

3）创建 nn 类的二维数组 nns。

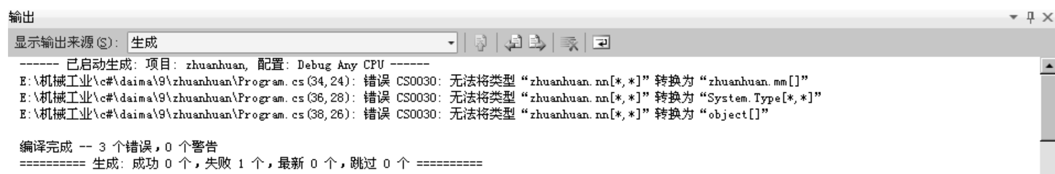
4）分别将 nns 转换为 mm 的一维数组和二维数组。

5）创建基类二维数组 aa 和 int 类型的二维数组 ints。

6）将 nns 转换为 Type 的二维数组。

7）将数组 ints 转换为值类型数组。

在 Visual Studio 2010 中运行上述代码后，将显示如图 9-8 所示的错误提示。


```
输出
显示输出来源(S): 生成
------ 已启动生成: 项目: zhuanhuan, 配置: Debug Any CPU ------
E:\机械工业\c#\daima\9\zhuanhuan\Program.cs(34,24): 错误 CS0030: 无法将类型“zhuanhuan.nn[*,*]”转换为“zhuanhuan.mm[]”
E:\机械工业\c#\daima\9\zhuanhuan\Program.cs(36,28): 错误 CS0030: 无法将类型“zhuanhuan.nn[*,*]”转换为“System.Type[*,*]”
E:\机械工业\c#\daima\9\zhuanhuan\Program.cs(38,26): 错误 CS0030: 无法将类型“zhuanhuan.nn[*,*]”转换为“object[]”

编译完成 -- 3 个错误，0 个警告
========== 生成: 成功 0 个，失败 1 个，最新 0 个，跳过 0 个 ==========
```

图 9-8　错误提示

从图 9-8 所示错误列表可以看出，上述代码中存在三处转换错误，具体如下。

1）“mm[] rcs2=(mm[])nns”，原因是两数组的维数不同。rcs2 是一维，nns 是二维。

2）“Type[,] types=(Type[,])nns”，原因是 Type 类型未知，不存在显式转换或隐式转换。

3）“object[] bb=(object[])nns”，原因是 ints 数组是值类型数组，不能被转换为其他类型。

9.3　C#集合类

在 System.Collections 命名空间内，定义了多个集合类实现特定的数据处理功能。常见的 System.Collections 集合类如下。

- ArrayList。
- Hashtable。
- Queue。
- SortedList。
- Stack。

9.3.1　ArrayList 集合类

C#数组的长度是固定的，应该在数组声明时就指定它的大小。通过使用 ArrayList 集合类，可以改变数组的长度，这样可以提高内存的利用率。

ArrayList 可以使用如下三种构造函数，这样能够根据需要而动态地增加数组。

1）ArrayList（），能够初始化 ArrayList 类的新实例，此实例为空并且具有默认初始容量。

2）ArrayList（ICollection），能够初始化 ArrayList 类的新实例，此实例包含从指定集合复制的元素，并且具有和所复制元素数相同的初始容量大小。

3）ArrayList（Int32），能够初始化 ArrayList 类的新实例，此实例为空并且指定初始容量。

ArrayList 类的常用属性如表 9-3 所示。

表 9-3　ArrayList 类属性

属　　性	描　　述
Capacity	获取设置 ArrayList 可以包含的元素数
Count	获取 ArrayList 实际包含的元素数
IsFixdSize	获取一个值，设置 ArrayList 是否具有固定大小
IsReadOnly	获取一个值，设置 ArrayList 是否为只读
IsSynchronized	获取一个值，设置是否对 ArrayList 的访问
Item	获取设置指定索引位置的元素
SyncRoot	获取可以同步 ArrayList 访问的对象

ArrayList 类的常用方法（函数）如表 9-4 所示。

表 9-4　ArrayList 类方法

方　法	描　述
Add()	将对象添加到 ArrayList 处
AddRange()	将 ICollection 的元素添加到 ArrayList 末尾
BinarySearch()	使用对分检索算法在已排序的 ArrayList 或它的一部分中查找特定元素
Clear()	从 ArrayList 中移除所有元素
Contains()	确定某元素是否在 ArrayList 内
Copyto()	将 ArrayList 或它的一部分复制到一维数组内
FixedSize()	返回具有固定大小的列表包装
GetEnumerator()	返回循环访问的 ArrayList 的列举器
GetRange()	返回 ArrayList，表示源 ArrayList 中元素的子集
IndexOf()	返回 ArrayList 或它一部分中某个值的第一个匹配项，并从 0 开始的索引
Insert()	将元素插入 ArrayList 内的指定索引处
InsertRange()	将集合中某元素插入 ArrayList 内的指定索引处
LastIndexOf()	返回 ArrayList 或某值的最后一个匹配项的索引
ReadOnly()	返回只读列表包装
Remove()	从 ArrayList 中移出特定对象的第一个匹配项
RemoveAt()	移出 ArrayList 的指定索引处的元素
RemoveRange()	移出 ArrayList 中一定范围的元素
Repeat()	返回 ArrayList
Reverse()	将 ArrayList 或它一部分的元素的顺序反转
SetRange()	将集合中的元素复制到 ArrayList 中的某范围的元素上
Sort()	对 ArrayList 的元素进行排序处理
Synchronized()	返回同步列表包装
ToArry()	将 ArrayList 复制到新数组中
TrimToSize()	将容量设置为 ArrayList 中元素的实际数目

其中，ArrayList 容量是 ArrayList 可以保存的元素数，其默认容量是 0。Add 能够向 ArrayList 中添加新的元素，通过添加后，容量会根据需要通过重新分配而自动增加容量。

下面将通过两个简单实例来说明 ArrayList 属性和方法的具体使用效果。

1．添加/删除数组数据

实例 34：通过 Add/Remove()方法添加/删除数组中的数据

下面将通过一个简单的实例来说明，使用 Add/Remove()方法实现数组数据的添加/删除处理的流程。本实例保存在“光盘：\daima\9”文件夹内，项目名为 adddel。本实例的功能是，输出显示指定数组内的数据，并将处理后的数组数据输出显示。实例文件的主要代码如下所示：

```
namespace adddel
{
   class Program
```

```
        {
            public static void Main()
            {
                ArrayList qiudui = new ArrayList();
                qiudui.Add("AA");
                qiudui.Add("BB");
                qiudui.Add("CC");
                qiudui.Add("DD");
                Console.WriteLine("我喜欢：{0}", qiudui.Count);
                Console.WriteLine("我最喜欢的数量：{0}", qiudui.Capacity);
                Console.WriteLine("我喜欢的包括：\n");
                for (int i = 0; i < qiudui.Count; ++i)
                    Console.WriteLine(qiudui[i]);
                if (qiudui.Contains("EE"))
                    Console.WriteLine("我喜欢的包括 EE。");
                else
                    Console.WriteLine("我喜欢的不包括 EE。");

                Console.WriteLine("删除 AA。\n");
                qiudui.Remove("AA");

                qiudui.Add("FF");
                qiudui.Add("GG");
                qiudui.Add("HH");
                Console.WriteLine("扩容后：\n");
                Console.WriteLine("数量为：{0}", qiudui.Count);
                Console.WriteLine("可容纳数量为：{0}\n", qiudui.Capacity);
                Console.WriteLine("我现在喜欢的包括：\n");
                for (int i = 0; i < qiudui.Count; ++i)
                    Console.WriteLine(qiudui[i]);
                    Console.ReadKey();
            }
        }
    }
```

上述实例代码的设计流程如下。

1）通过 using 调用 System.Collections 类。

2）定义 adddel 命名空间和 Program 类。

3）定义数组 qiudui，通过方法 Add()添加 4 个初始成员。

4）输出数组的原始成员数和对应的成员数据。

5）通过 Remove()方法删除数组内的成员“AA”。

6）通过方法 Add()为数组添加另外 3 个成员。

7）将数组扩容后的数据输出显示。

执行后将输出添加处理后的原始数组数据，然后显示删除后的数据，最后显示扩容后的

数据，具体如图 9-9 所示。

```
我喜欢：4
我最喜欢的数量：4
我喜欢的包括：

AA
BB
CC
DD
我喜欢的不包括EE。
删除AA。

扩容后：

数量为：6
可容纳数量为：8

我现在喜欢的包括：

BB
CC
DD
FF
GG
HH
```

图 9-9　实例执行结果

2．第三种构造函数确定处理

第三种构造函数即确定最大容量的构造函数。从图 9-9 所示的现实结果可以看出，随着 ArrayList 内成员的增加，Capacity 值也会随之增加。这样就需要不断扩展 Capacity 容量并进行元素复制，所以会影响程序的执行效率。为解决上述问题，可以使用第三种形式的构造函数来创建 ArrayList，并在其中设置一个最大容量值。只要元素数不超过这个最大值，ArrayList 就不需要再进行容量扩展和复制，这样就提高了程序的效率。

9.3.2　Hashtable 集合类

Hashtable 是表示键值对的集合，键值能够根据键的散列代码进行组织。Hashtable 类的常用属性如表 9-5 所示。

表 9-5　Hashtable 类的常用属性

属　　性	描　　述
Count	获取 Hashtablet 内的键值数
IsFixdSize	获取一个值，设置 Hashtablet 是否具有固定大小
IsReadOnly	获取一个值，设置 Hashtablet 是否为只读
IsSynchronized	获取一个值，设置是否对 Hashtablet 的访问
Item	获取设置与指定键相关联的元素
SyncRoot	获取可以同步 Hashtablet 访问的对象
Keys	获取包含 Hashtablet 中的键的 ICollection
Value	获取包含 Hashtablet 中的值的 ICollection

Hashtable 类的常用方法（函数）如表 9-6 所示。

表 9-6　Hashtable 类的常用方法

方　　法	描　　述
Add()	将指定键值元素添加到 Hashtablet
Clear()	从 Hashtablet 中移除所有元素
Contains()	确定 Hashtablet 内是否包含特定键
Clone()	创建 Hashtablet 的浅表副本
Copyto()	将 Hashtablet 元素复制到一维 ArrayList 实例中的指定索引位置
ContainsKey()	确定 Hashtablet 内是否包含特定键
ContainsValue()	确定 Hashtablet 内是否包含特定键
GetEnumerator()	返回循环访问 Hashtablet 的列举器
GetObjectData()	实现 ISerializable 接口，并返回序列化 Hashtablet 所需要的值
OnDeserialization()	实现 ISerializable 接口，并在完成反序列化之后引发反序列化事件
Remove()	移出 Hashtablet 内指定键的元素
RemoveRange()	移出 ArrayList 中一定范围的元素
Synchronized()	返回 Hashtablet 的同步包装

其中，Add()能够将键值添加到 Hashtablet 中，并且 Hashtablet 中的值不能为空引用。

Hashtablet 将添加元素放在 System.Collections.DictionaryEnty 对象中。所以在使用 foreach 或 IEnumerator 的 Current 属性时，应该将 Hashtablet 元素转换为 DictionaryEnty 对象。

使用 Remove()能够删除指定键值的元素，使用 Clear()能够删除所有的元素。

Hashtablet 的默认初始值为 0。随着新元素的添加，容量会根据需要通过重新分配后自动增加。并且需要重新计算表的索引，并将表中的每个元素复制到新的位置。

下面将通过两个简单实例来说明使用 ArrayList 属性和方法的效果。

1．添加/删除数组数据

实例 35：通过 Add/Remove()方法实现数组数据的添加/删除处理

下面将通过一个简单的实例来说明，通过 Add/Remove()方法实现数组数据的添加/删除处理。本实例保存在“光盘：\daima\9”文件夹内，项目名为 add。本实例的功能是，输出显示处理后的数组数据。实例文件的主要代码如下所示：

```
namespace add
{
   public class Test
   {
      public static void Main()
      {
         Hashtable mm = new Hashtable();
         mm.Add("Sunday", "星期天");
         mm.Add("Monday", "星期一");
         mm.Add("Tuesday", "星期二");
         mm["Wednesday"] = "星期三";
         Console.WriteLine("这周可以休息{0}天", mm.Count);
         Console.WriteLine("Tuesday 对应的是：{0}", mm["Tuesday"]);
```

```
                Console.WriteLine("这周可以休息的是：\n");
                IEnumerator e = mm.GetEnumerator();
                while (e.MoveNext())
                {
                    DictionaryEntry de = (DictionaryEntry)e.Current;
                    Console.WriteLine("{0}-{1}", de.Key, de.Value);
                }
                if (mm.ContainsKey("Wednesday"))
                    Console.WriteLine("\n 已经包含了 Wednesday。");
                else
                    Console.WriteLine("没有包含 Wednesday。");
                if (mm.ContainsValue("星期三"))
                    Console.WriteLine("已经包含了星期三。");
                else
                    Console.WriteLine("没有包含星期三。");
                Console.WriteLine("\n 删除 Wednesday-星期三。");
                mm.Remove("Wednesday");
                mm.Add("Thursday", "星期四");
                mm.Add("Friday", "星期五");
                mm.Add("Saturday", "星期六");
                Console.WriteLine("\n 增加休息天数：");
                Console.WriteLine("可以休息{0}天", mm.Count);
                Console.WriteLine("这周可以休息的是：\n");
                Console.ReadKey();
                e = mm.GetEnumerator();
                while (e.MoveNext())
                {
                    DictionaryEntry de = (DictionaryEntry)e.Current;
                    Console.WriteLine("{0}-{1}", de.Key, de.Value);
                }
                Console.ReadKey();
            }
        }
    }
```

上述实例代码的设计流程如下。

1）通过 using 调用 System.Collections 类。

2）定义命名空间 add 和 Test 类。

3）通过集合类初始化 mm，然后通过 Add()方法向 mm 内添加 3 个键值。其中英文字符格式是键，中文字符格式是值。

4）输出显示添加处理后的数据。

5）通过 Remove()方法删除指定“Wednesday”键的值。

6）输出显示删除处理后的数据。

7）通过 Add()方法向 mm 内添加另外 3 个键值，其中英文字符格式是键，中文字符格式是值。

8）输出显示第二次添加处理后的数据。

执行后首先输出显示第一次添加/删除处理后的列表数据，如图 9-10 所示。当按〈Enter〉键后将输出显示第二次添加处理后的数据，如图 9-11 所示。

```
这周可以休息4天
Tuesday对应的是：星期二
这周可以休息的是：

Monday-星期一
Sunday-星期天
Tuesday-星期二
Wednesday-星期三

已经包含了Wednesday。
已经包含了星期三。

删除Wednesday-星期三。

增加休息天数：
可以休息6天
这周可以休息的是：
```

图 9-10　实例执行结果

```
Saturday-星期六
Monday-星期一
Sunday-星期天
Friday-星期五
Tuesday-星期二
Thursday-星期四
```

图 9-11　实例执行结果

2．键值用做索引处理

用键值实现索引处理和使用 Add()方法进行添加处理是有区别的，具体说明如下。

1）添加的元素键值在数组中存在时，Add()方法会引发异常。

2）当使用键值作为索引时，如果添加的元素键值在数组中存在时不会发生异常，它会取代原来的值。

9.3.3　Queue 集合类

Queue 是一个表示对象的先进先出的集合，通常用于按接收顺序存储信息领域，以便更好地实现顺序处理。Queue 是通过循环数组来实现的，里面的对象总是从一端插入，而从另一端移出。

Queue 类的常用属性如表 9-7 所示。

表 9-7　Queue 类属性

属　　性	描　　述
Count	获取 Queue 内的元素数
IsSynchronized	获取一个值，设置是否同步对 Queue 的访问
SyncRoot	获取可以同步 Queue 访问的对象

Queue 类常用方法（函数）如表 9-8 所示。

表 9-8　Queue 类方法

方　　法	描　　述
Clear()	从 Queue 中移除所有元素
Contains()	确定 Queue 内是否包含某元素
Clone()	创建 Queue 的浅表副本
Copyto()	从指定数组索引开始将 Queue 元素复制到现有一维 ArrayList 中

（续）

方　　法	描　　述
Dequeue()	移除并返回位于 Queue 开始处的对象
Enqueue()	将对象添加到 Queue 的结尾处
GetEnumerator()	返回循环访问 Queue 的列举器
Peek()	返回 Queue 开始处的对象
ToArray()	将 Queue 元素复制到新的数组内
Synchronized()	返回 Queue 的同步包装

其中，Queue 的容量是它能够保存的元素数量，其默认初始容量为 32。当向 Queue 添加新元素时，通过重新分配根据需要来自动增大容量，通过 TrimToSize 来减小容量。

实例 36：使用 Queue 处理列表数据

下面将通过一个简单的实例来说明使用 Queue 的流程。本实例保存在“光盘：\daima\9”文件夹内，项目名为 queueshiyong。本实例的功能是，向列表内添加数据，并将处理后的列表数据输出显示。实例文件的主要代码如下：

```
namespace queueshiyong {
public class Test {
   public static void Main()
   {
      // 创建并初始化 Queue
      Queue mm = new Queue();
      mm.Enqueue("AA");
      mm.Enqueue("BB");
      mm.Enqueue("CC");
      // 显示 Queue 的信息
      Console.WriteLine("里面的元素数量: {0}", mm.Count);
      Console.WriteLine("里面的内容：");
      IEnumerator e = mm.GetEnumerator();
      while (e.MoveNext())
      {
         Console.WriteLine(e.Current);
      }
      Console.ReadKey();
   }
}
}
```

上述实例代码的设计流程如下。

1）通过 using 调用 System.Collections 类。

2）定义命名空间 queueshiyong 和 Test 类。

3）通过 Queue 集合类初始化 mm，然后通过方法 Enqueue()向 mm 内添加 3 个元素。

4）通过属性 Count 输出数组内的数据个数。

5）通过 while 语句获取当前数组内的数据。

6）输出显示列表内的当前数据。

执行后将输出显示添加处理后的列表数据，如图 9-12 所示。

在使用 Queue 集合类时，应该特别注意它的先进先出特性。

```
里面的元素数量：3
里面的内容：
AA
BB
CC
```

图 9-12　实例执行结果



多学一招

在 C#中，当使用 Queue 集合类时应该特别注意它的先进先出特性。Queue 和 ArrayList 集合类比较类似，在此笔者对两者进行简要总结。

1）都是集合。

2）都使用 Count()属性返回当前集合成员数。

3）先进先出原则。

4）Arraylist 用 Capacity()属性返回容量，随着成员数的 add，Capacity()值会动态增加。

5）Queue 没有 Capacity()属性，是个堆栈，没有固定的容量。

6）IEnumerable 可以使集合类型有 foreach()功能，它是一个泛型接口。如 IEnumerrable arraylist 相当于 Arraylist arraylist，但是前者的 arraylist 就具备了 foreach()方法。

9.3.4　SortedList 集合类

SortedList 是表示键值对的集合，通过这些键值可以对按键进行排序，并且可以按照按键和索引进行访问处理。在具体使用上，SortedList 和 Hashtable 极其类似。

SortedList 类的常用属性如表 9-9 所示。

表 9-9　SortedList 类的常用属性

属　　性	描　　述
Capacity	设置 SortedList 容量
Count	获取 SortedList 内的元素数
IsFixdSize	获取一个值，设置 SortedList 是否具有固定大小
IsReadOnly	获取一个值，设置 SortedList 是否为只读
IsSynchronized	获取一个值，设置是否同步对 SortedList 的访问
Item	获取设置与 SortedList 中指定键相关联的值
SyncRoot	获取可以同步 SortedList 访问的对象
Keys	获取 SortedList 中的键
Value	获取 SortedList 中的值

SortedList 类的常用方法（函数）如表 9-10 所示。

表 9-10　SortedList 类的常用方法

方　　法	描　　述
Add()	将指定键值元素添加到 SortedList
Clear()	从 SortedList 中移除所有元素
Contains()	确定 SortedList 是否包含特定键
Clone()	创建 SortedList 的浅表副本

（续）

方　　法	描　　述
Copyto()	将 SortedList 元素复制到一维 ArrayList 实例中的指定索引位置
ContainsKey()	确定 SortedList 内是否包含特定键
GetByIndex()	获取 SortedList 指定索引处的值
ContainsValue()	确定 SortedList 内是否包含特定键
GetEnumerator()	返回循环访问 SortedList 的列举器
GetKey()	获取 SortedList 指定索引处的键
GetKeyList()	获取 SortedList 中的键
GetValueList()	返回 SortedList 中的值
IndexOfKey()	返回 SortedList 中指定键的从 0 开始的索引
IndexOfValue()	返回指定值在 SortedList 中第一个匹配项从 0 开始的索引
Remove()	移出 SortedList 内指定键的元素
RemoveAt()	移出 SortedList 内指定索引处的元素
SetByIndex()	替换 SortedList 中指定索引处的值
Synchronized()	返回 SortedList 的同步包装
TimeToSize()	将容量设置为 SortedList 内元素的实际数

其中，可以通过键或值来访问 SortedList 内的元素。在 SortedList 内将维护两个数组来存储列表内元素，其中将一个数组存储键，另一个数组用于相对应的值。

SortedList 的容量是 SortedList 可以保存的元素数量，默认值为 0。当把新元素添加到 SortedList 后，将根据需要而自动增加容量。可以通过 TimeToSize 和设置 Capacity()属性来改变容量。



SortedList 需要进行排序处理，所以用 SortedList 处理的速度要慢于 Hashtable 的处理。但是 Capacity 允许通过关联键或索引对值进行访问，从而提供更大的灵活性。

实例 37：使用 SortedList 集合类处理数组数据

下面将通过一个简单的实例来说明使用 SortedList 集合类的流程。本实例保存在“光盘：\daima\9”文件夹内，项目名为 SortedListjihe。本实例的功能是，向数组内添加数据，并输出显示处理后的数组数据。实例文件的主要代码如下：

```
namespace SortedListjihe {
public class Test {
   public static void Main()
   {
      // 创建并初始化 nn
      SortedList nn = new SortedList();
      nn.Add("5", "曼联");
      nn.Add("4", "切尔西");
      nn.Add("1", "利物浦");
      nn.Add("3", "阿森纳");
      nn.Add("2", "罗马");
```

```
            nn.Add("6", "巴塞罗那");
            nn.Add("7", "费内巴切");
            nn.Add("8", "沙尔克04");
            // 显示nn的信息
            Console.WriteLine("08赛季欧洲冠军杯{0}强", nn.Count);
            Console.WriteLine("球队：");
            for (int i = 0; i < nn.Count; i++)
            {
                Console.WriteLine("\t{0}:\t{1}", nn.GetKey(i), nn.GetByIndex(i));
            }
            Console.ReadKey();
        }
    }
    }
```

上述实例代码的设计流程如下。

1）通过 using 调用 System.Collections 类。

2）定义命名空间 SortedListjihe 和 Test 类。

3）通过 SortedList 集合类初始化 nn，然后通过 Add()方法向 nn 内添加八个键值。其中键为数字格式，值为文本格式。

4）通过 for 语句获取当前数组内的数据。

5）输出显示数组内的当前数据。

执行后将输出显示添加处理后的数组数据，如图 9-13 所示。

```
08赛季欧洲冠军杯8强
球队:
        1:      利物浦
        2:      罗马
        3:      阿森纳
        4:      切尔西
        5:      曼联
        6:      巴塞罗那
        7:      费内巴切
        8:      沙尔克04
```

图 9-13　实例执行结果

从图 9-13 所示结果可以看出，虽然添加时的顺序是无序的，但是添加到 nn 后，会根据键的序号进行排序处理。


多学一招

因为 SortedList 需要进行排序处理，所以用 SortedList 处理的速度要慢于 Hashtable 的处理。但是 Capacity 允许通过关联键或索引对值进行访问，从而提供更大的灵活性。

SortedList 是 Hashtable 和 Array 的结合体。当使用 Item 索引器属性按照元素的键访问元素时，其行为类似于 Hashtable。当使用 GetByIndex 或 SetByIndex 按照元素的索引访问元素时，其行为类似于 Array。SortedList 在内部维护两个数组以将数组存储到列表中；即一个数组用于键，另一个数组用于相关联的值。每个元素都是一个可作为 DictionaryEntry 对象进行

访问的键/值对。键不能为空引用（Visual Basic 中为 Nothing），但值可以。

SortedList 的容量是列表可拥有的元素数。当向 SortedList 中添加元素时，容量通过重新分配按需自动增加。可通过调用 TrimToSize 或通过显式设置 Capacity()属性来减小容量。

9.3.5　Stack 集合类

Stack 是一种表示对象后进先出的集合，通常用于按接收的反向顺序存储信息，以便更好地实现顺序处理。Stack 是通过循环数组来实现的，里面的对象总是从一端插入，而从另一端移出。Stack 类的常用属性如表 9-11 所示。

表 9-11　Stack 类的常用属性

属　　性	描　　述
Count	获取 Stack 内的元素数
IsSynchronized	获取一个值，设置是否同步对 Stack 的访问
SyncRoot	获取可以同步 Stack 访问的对象

Stack 类的常用方法如表 9-12 所示。

表 9-12　Stack 类的常用方法

方　　法	描　　述
Clear()	从 Stack 中移除所有元素
Copyto()	从指定数组索引开始将 Stack 元素复制到现有一维 Array 中
GetEnumerator()	返回循环访问 Stack 的列举器
Peek()	返回 Stack 顶部的对象
Pop()	删除并返回位于 Stack 顶部的对象
Push()	将对象插入 Stack 顶部
ToArray()	将 Stack 元素复制到新的数组内
Synchronized()	返回 Stack 的同步包装

Stack 是通过循环来实现的，它的容量是 Stack 可以保存的元素数。Stack 的默认初始容量值为 10，当向 Stack 内添加新元素时，将根据需要而重新分配，实现自动增加容量。

实例 38：使用 Stack 处理数组数据

下面将通过一个简单的实例来说明使用 Stack 的流程。本实例保存在“光盘：\daima\9”文件夹内，项目名为 Stackshiyong。本实例的功能是向数组内添加数据，并将处理后的数组数据输出显示。实例文件的主要代码如下：

```
namespace Stackshiyong {
   public class Test {
      public static void Main()
      {
         // 创建并初始化 nn
         Stack nn = new Stack();
         nn.Push("AA");
```

```
                nn.Push("BB");
                nn.Push("CC");
                nn.Push("DD");
                nn.Push("EE");
                nn.Push("FF");
                nn.Push("GG");
                nn.Push("HH");

                // 显示nn的信息
                Console.WriteLine("08赛季欧洲冠军杯{0}强俱乐部: ", nn.Count);
                IEnumerator e=nn.GetEnumerator();
                while (e.MoveNext())
                {
                    Console.WriteLine(e.Current);
                }
                Console.ReadKey();
            }
        }
    }
```

上述实例代码的设计流程如下。

1）通过 using 调用 System.Collections 类。

2）定义命名空间 Stackshiyong 和 Test 类。

3）通过 Stack 集合类初始化 nn，然后通过方法 Push()向 nn 内添加 8 个元素。

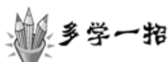
4）通过 while 语句获取当前数组内的数据。

5）输出显示数组内的当前数据。

执行后将输出显示添加处理后的列表数据，如图 9-14 所示。

```
08赛季欧洲冠军杯8强俱乐部:
HH
GG
FF
EE
DD
CC
BB
AA
```

图 9-14　实例执行结果



Stack 的特点是先进后出，后进先出，看下面的代码:

```
Stack p = new Statck();
p.Push(1);
p.Push(2);
int s1 = (int)p.Pop();
int s2 = (int)p.Pop();
```

上述代码执行后，s1 的值应该是 2, s2 的值应该是 1。

Stack 的容量是指 Stack 可以保存的元素数。在实现中 Stack 的默认初始容量为 10；但此默认值可能在 .NET Framework SDK 的未来版本中更改。向 Stack 添加元素时，将通过重新分配内部数组，根据需要自动增大容量。可通过调用 TrimExcess 来减小容量。

如果 Count 小于堆栈的容量，则 Push 为 O(1) 操作。如果需要增加容量以容纳新元素，则 Push 为 O(n)操作，其中 n 为 Count。Pop 为 O(1)操作。Stack 接受空引用（在 Visual Basic 中为 Nothing） 作为引用类型的有效值并且允许有重复的元素。

9.4　使用 foreach 循环语句

在本书的第 5 章中，已经详细介绍了 C#循环语句的基本知识。在本节的内容中，将结合集合的基本知识，向读者简要介绍 foreach 循环语句的使用流程。

9.4.1　foreach 概述

在 C#集合内，foreach 循环可以隐藏集合类型的内部实现方法，从而更加有效地处理集合类型元素。通过 foreach 循环语句，可以循环列举某集合内的元素，并对各元素执行一次相关的嵌入语句。

foreach 的具体使用格式如下：

```
foreach(类型 迭代变量) in 表达式
   {
     处理语句
   }
```

其中，通过类型和迭代变量来声明语句的迭代变量。C#中的迭代变量，相当于某范围内覆盖整个嵌入语句的只读局部变量。当在 foreach 语句执行时，迭代变量表示当前正在为其执行迭代的集合元素。



1）如果运行 foreach 处理语句试图修改迭代变量值，或将变量值作为 ref 参数或 out 参数传递，都会发生编译错误。

2）foreach 语句的表达式必须有一个从该集合的元素类型到迭代类型的显式转换，如果表达式的结果为 null，则会发生运行异常。

3）如果在某集合内支持 foreach 语句，则这个集合必须能够实现 System.Collections.IEnumerable 接口，或者实现集合模式。实现集合模式的条件如下。

❑ 该集合包含一个 public 实例方法，这个方法含有签名 GetEnumerator（），并且其返回值为结构、类或接口类型。

❑ 该集合包含一个 public 实例方法，这个方法含有签名 MoveNext（）和返回类型 bool。

❑ 该集合包含一个 public 类型的 Current 属性实例，这个属性允许读取当前的值。

如果表达式的集合类型实现了集合模式，则可以通过如下格式使用 foreach 语句：

```
集合类 enumerator = (collection). GetEnumerator();
try{
ElementType element;
while (enumerator.MoveNext())
   {
     element=(ElementType) enumerator. Current;
     处理语句;
}
```

```
}
finally
{
IDisposable disposable= enumerator as System. IDisposable;
i(disposable!=null)
disposable. Dispose();
}
```

9.4.2　集合内使用 foreach 语句

在命名空间 System.Collections 中，集合类型 ArrayList、Hashtable、Queue、SortedList 和 Stack 都支持 foreach 语句。在下面的内容中，将通过具体的实例来演示 foreach 在集合类型中的应用过程。

实例 39：在集合类型中应用 foreach

下面首先通过一个简单实例的实现过程，来说明在集合类型中使用 foreach 语句的流程。本实例保存在“光盘：\daima\9”文件夹内，项目名为 foreach。本实例的功能是，向列表内添加数据，并输出显示处理后的列表数据。实例文件的主要代码如下：

```
namespace foreach1
{
    public class Test
    {
        public static void Main()
        {
            ArrayList lib = new ArrayList();
            lib.Add("AA");
            lib.Add("BB");
            lib.Add("CC");
            lib.Add("DD");
            lib.Add("EE");
            Console.WriteLine("我的最爱：");
            foreach (string book in lib)
            {
                Console.WriteLine(book);
            }
            double[,] values =
        {
                {1.2, 2.3, 3.4, 4.5},
                {5.6, 6.7, 7.8, 8.9}
          };
            Console.WriteLine("\n 数组的值为：");
            foreach (double elementValue in values)
                Console.Write("{0} ", elementValue);
                Console.ReadKey();
        }
```

```
        }
    }
```

上述实例代码的设计流程如下。

1）通过 using 调用 System.Collections 类。

2）定义命名空间 foreach1 和 Test 类。

3）通过 ArrayList 集合类初始化 lib，然后通过方法 Add()向 lib 内添加 5 个元素。

4）通过 foreach 语句获取当前数组内的数据。

5）输出显示数组内的当前数据。

执行后将输出显示添加处理后的数组数据，如图 9-15 所示。

```
我的最爱：
AA
BB
CC
DD
EE

数组的值为：
1.2 2.3 3.4 4.5 5.6 6.7 7.8 8.9
```

图 9-15　实例执行结果

9.5　自定义集合类

除了本章前面介绍的常用集合类外 ，还可以根据现实需要自行定义集合类。在本节的内容中，将向读者详细介绍 C#中自定义集合类的方法。

9.5.1　实现接口

在 C#中，必须先实现 System.Collections.IEnumerabl 和 System.Collections. IEnumerator 的接口，然后才能使用自定义集合类。因为方法 IEnumerator()会返回一个 IEnumerator 对象，所以需要编写一个实现了 IEnumerator 接口的 IEnumerator 类。即在方法 IEnumerator()中，创建这个类的对象并返回即可。

通常将自定义类声明为内部类，这是因为创建的 IEnumerator 类只需要在自定义的集合类内使用。其中，IEnumerator 类必须实现 IEnumerator 接口的两个方法，即 Rset()和 MoveNext()，还有属性 Current。

实例 40：自定义创建集合类 mm

下面将通过一个简单实例来说明自定义创建一个名为 mm 集合类的过程。本实例保存在“光盘：\daima\9”文件夹内，项目名为 zidingyi。本实例的功能是，向列表内添加数据，并输出显示处理后的列表数据。实例文件的主要代码如下：

```
namespace zidingyi {
public class mm : IEnumerable {
   protected class nn : IEnumerator {
      private string exceptionInfo = "索引无效";
      private int index = -1;
      private ArrayList qiuduis = new ArrayList();
      public nn(ArrayList qiuduis)
      {
         foreach (string qiudui in qiuduis)
         {
            this.qiuduis.Add(qiudui);
         }
```

```
                Reset();
            }
            public bool MoveNext()
            {
                return (++index < qiuduis.Count);
            }
            public void Reset()
            {
                index = -1;
            }
            public object Current
            {
                get
                {
                    if (0 <= index && index < qiuduis.Count)
                        return qiuduis[index];        // 返回当前元素
                    else                              // 索引无效
                        throw new IndexOutOfRangeException(exceptionInfo);
// 抛出异常
                }
            }
        }
        protected ArrayList qiuduis;
        public mm()
        {
            qiuduis = new ArrayList();
        }
        public void Add(string qiudui)
        {
            qiuduis.Add(qiudui);
        }
        public void Remove(string qiudui)
        {
            qiuduis.Remove(qiudui);
        }
        public IEnumerator GetEnumerator()
        {
            return new nn(qiuduis);
        }
    }
    public class Test {
        public static void Main() {
            mm club = new mm();
            club.Add("ＡＡ");
            club.Add("ＢＢ");
            club.Add("ＣＣ");
```

```
                club.Add("ＤＤ");
                club.Add("ＥＥ");
                Console.WriteLine("我喜欢的有：\n");
                foreach (string qiudui in club)
                {
                    Console.WriteLine(qiudui+"\n");
                }
                Console.ReadKey();
            }
        }
        }
```

上述实例代码的设计流程如下。

1）通过 using 调用 System.Collections 类，定义命名空间 zidingyi。

2）定义类 mm，实现和 IEnumerabl 的接口。

3）定义类 nn，实现和 IEnumerator 的接口。

4）设置变量 exceptionInfo 和 index 的初始值。

5）定义 ArrayList 类型成员 qiuduis，用于存储元素。

6）分别实现方法 Rset()、MoveNext()和属性 Current 的处理。

7）定义自定义类 mm 的成员 club，用于存储球队。

8）通过方法 Add()向 club 里面添加五个数据。

9）使用 foreach 将当前的元素数据输出显示。

上述实例代码执行后，将输出显示添加处理后的元素数据，具体如图 9-16 所示。

9.5.2　结合实现

可以同时实现接口 IEnumerabl 和 IEnumerator，实现方法很简单，只需让自定义的集合类实现 IEnumerator 接口，然后在类内实现方法 Rset()和 MoveNext()，还有属性 Current 即可，这样大大减少了代码的编写量。

```
我喜欢的有：
AA
BB
CC
DD
EE
```

图 9-16　实例执行结果

实例 41：结合实现自定义集合类 mm

下面将通过一个简单实例来说明结合实现自定义集合类 mm 的过程。本实例保存在“光盘：\daima\9”文件夹内，项目名为 jiehezidingyi。本实例的功能是，向列表内添加数据，并输出显示处理后的列表数据。实例文件 Program.cs 的主要代码如下：

```
        namespace jiehezidingyi {
        public class mm : IEnumerable, IEnumerator {
            private string aa = "索引无效";
            private int index = -1;
            protected ArrayList qiuduis;
            public mm()
            {
                qiuduis = new ArrayList();
            }
```

```
        public void Add(string qiudui)
        {
            qiuduis.Add(qiudui);
        }
        public void Remove(string qiudui)
        {
            qiuduis.Remove(qiudui);
        }
        public IEnumerator GetEnumerator()
        {
            Reset();
            return this;
        }
        public bool MoveNext()
        {
            return (++index < qiuduis.Count);
        }
        public void Reset()
        {
            index = -1;
        }
        public object Current
        {
            get
            {
                if (0 <= index && index < qiuduis.Count)
                    return qiuduis[index];                     // 返回当前元素
                else                                           // 索引无效
                    throw new IndexOutOfRangeException(aa);    // 抛出异常
            }
        }
    }
    public class Test {
        public static void Main() {
            mm club = new mm();
            club.Add("ＡＡ");
            club.Add("ＢＢ");
            club.Add("ＣＣ");
            club.Add("ＤＤ");
            club.Add("E E");
            Console.WriteLine("我喜欢的俱乐部有：\n");
            foreach (string qiudui in club)
            {
                Console.WriteLine(qiudui + "\n");
            }
```

```
            Console.ReadKey();
        }
    }
    }
```

上述实例代码的设计流程如下。

1）通过 using 调用 System.Collections 类，定义命名空间 jiehezidingyi。

2）定义类 mm，设置同时实现和 IEnumerabl、IEnumerator 的接口。

3）设置变量 aa 和 index 的初始值。

4）如果当前索引无效则输出变量 aa 的错误提示。

5）定义类 mm 的构造函数，并设置其 ArrayList 类型成员 qiuduis，用于存储球队。

6）分别实现方法 Rset()、MoveNext()和 GetEnumerator 的处理。

7）定义自定义类 mm 的成员 club，用于存储俱乐部。

8）通过方法 Add()向 club 里面添加 5 个数据。

9）使用 foreach 将当前的球队数据输出显示。

执行后将输出显示添加处理后的俱乐部数据，具体如图 9-17 所示。

```
我喜欢的俱乐部有：
AA
BB
CC
DD
EE
```

图 9-17　实例执行结果

9.6　C#迭代器

如果在 C#中使用 foreach 语句，必须先在迭代的类中实现 IEnumerabl 接口或 ICollection 接口。上述处理过程实现起来会比较麻烦，为解决这个问题，在 C#中引入了一种名为迭代器的概念。只要在 C#项目程序的类或结构中实现迭代器，那么这个类和结构就支持 foreach 迭代，而不再需要实现全部的 IEnumerabl 接口。当编译器检测到迭代器时，项目程序将自动实现 IEnumerabl 接口或实现 IEnumerabl 接口的方法和属性。

迭代器能够生成相同类型的有序代码块，通过上述代码块能够实现特定的处理功能。迭代器代码块和普通代码块类似，只是在迭代器代码块里存在不定量的 yield 语句。具体使用格式如下：

```
foreach 语句　表达式
  {
    yield return 成员;
    ……
    yield break;
  }
```

其中，yield return 语句生成迭代的下一个值；yield break 语句设置迭代终止完成。即关键字 yield 用于设置返回值，当程序达到 yield return 语句时，会保存当前的位置。当下次调用迭代器时，将从这个位置重新开始执行。

可以在一个类中实现多个迭代器，但是其中的每个迭代器都必须具有唯一的名称，并能够在 foreach 语句中被客户端调用。



迭代器不是一种成员，而是一种实现函数成员的方法。迭代器可以作为函数、运算符或 get 访问器的代码体。通过使用迭代器成员，可以被其他成员实现重写或重载。

迭代器在集合类中的应用比较常见，它能够提供一种简单的方法来迭代不常用的数据结构。

实例 42：在集合类中应用迭代器

下面将通过一个简单实例来说明在集合类中使用迭代器的过程。本实例保存在“光盘：\daima\9”文件夹内，项目名为 diedaiqi。本实例的功能是，输出显示集合类内的元素。实例文件的主要代码如下：

```
namespace diedaiqi {
public class DaysOfTheWeek : System.Collections.IEnumerable {
    string[] m_Days = { "天", "一", "二", "三", "四", "五", "六" };
    public System.Collections.IEnumerator GetEnumerator()
    {
        for (int i = 0; i < m_Days.Length; i++)
        {
            yield return m_Days[i];
        }
    }
}
class Test
{
    static void Main()
    {
        DaysOfTheWeek week = new DaysOfTheWeek();
        foreach (string day in week)
        {
            System.Console.Write("星期{0}\n" ,day );
        }
        Console.ReadKey();
    }
}
}
```

上述实例代码的设计流程如下。

1）通过 using 调用 System.Collections 类，定义命名空间 diedaiqi。

2）定义 IEnumerabl 接口的类 DaysOfTheWeek。

3）定义数组 m_Days，并设置存储七个成员值。

4）通过迭代逐一遍历数组 m_Days 存储的成员数据。

5）定义类 DaysOfTheWeek 对象 Week。

6）使用 foreach 将数组内数据输出显示。

执行后将输出显示数组内的成员数据，如图 9-18 所示。



多学一招

在使用迭代器时应该注意如下几点。

1）迭代器不是一种成员，而是一种实现函数成员的方法。迭代器可以作为函数、运算符或 get 访问器的代码体。通过使用迭代器成员，可以被其他成员实现重写或重载。

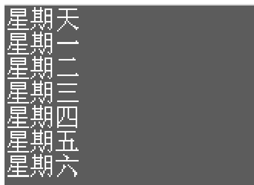


图 9-18　实例执行结果

2）可以在类中实现多个迭代器。每个迭代器都必须像任何类成员一样有唯一的名称，并且可以在 foreach 语句中被客户端代码调用，例如：foreach(int x in SampleClass.Iterator2){}。

3）迭代器的返回类型必须为 IEnumerable、IEnumerator、IEnumerable<(Of <(T>)>)或 IEnumerator<(Of <(T>)>)。

4）yield 关键字用于指定返回的一个或多个值。到达 yield return 语句时，会保存当前位置。下次调用迭代器时将从此位置重新开始执行。

使用迭代器的基本原则是，如果要迭代一个类则使用 GetEnumerator()，返回类型是 Enumerator。如果要迭代一个类成员，则使用 Enumerable。

9.7　疑难问题解析

本章详细介绍了 C#中集合和数组的基本知识。本节将对本章中比较难以理解的问题进行讲解。

读者疑问：请问 C#集合可以扩展吗？

解答：可以扩展，在扩展时可以借鉴很多数学应用软件，例如 MatLab。假如在应用中需要验证一个字符串是否为合法，一般的写法是定义一个类，并实现静态方法 IsValid(string):

```
string email = Request.QueryString["email"];
if ( EmailValidator.IsValid(email) ) {
}
```

而扩展方法则循序用下面的格式:

```
string email = Request.QueryString["email"];
if ( email.IsValidEmailAddress() ) {
}
```

使用扩展方法的实现代码如下:

```
public static class ScottGuExtensions
{
   public static bool IsValidEmailAddress(this string s)
   {
      Regex regex = new Regex(@"^[/w-/.]+@([/w-]+/.)+[/w-]{2,4}$");
```

```
            return regex.IsMatch(s);
        }
    }
```

利用扩展方法这个新特性来给个别类型添加方法，给开发人员开辟了许多有用的扩展性使用场景。但使得扩展方法非常强有力的是，它们不仅能够应用到个别类型上，也能应用到.NET 框架中任何基类或接口上。这允许开发人员建立种种可用于整个.NET 框架的丰富的可组合的框架层扩展。



职场点拨——老板喜欢什么样的员工

在职场中大家需要面对各种关系，例如客户关系、同事关系、上下级关系。在这些关系中，直接关系到前途的是和上级的关系，所以处理好和上级的关系，能够促进我们的发展。要处理好和上级的关系，需要先了解老板们喜欢什么样的员工，希望下面的内容或许能够为读者带来启示。

（1）低调而忠诚

绝大部分老板们都喜欢对公司忠诚、对外低调的员工。要想长期追随老板，必须先获得他们的信任，无论遇到什么事情，都要先为老板考虑。要想赢得老板对自己的信任，做事必须低调而沉稳。

（2）积极主动

在领导面前一定要积极主动，当然这种积极主动不是做一些额外的琐事，而是在分内的工作之外做事。例如程序员在完成自己的编码任务之后，在征得同事的同意下，主动向领导请缨，帮助同事完成任务。

（3）人脉网络充沛，高度自我管理

要被领导赏识，还需要具有充沛的人脉网络，并能在工作中自我管理，确保高水平的工作表现。充沛的人脉要求能够透过信息交换，与公司以外的专业人士建立起彼此信赖的沟通管道，以减少在工作中碰到的知识盲点。这个以专业知识为主轴建立起的人脉网络，可让明星员工比同僚更迅速地掌握信息，提高生产力。

（4）视野宏观、远瞻

要具备宏观的视野，要有普通员工所没有的前瞻性。例如在面对一个项目时，要充分考虑到以后系统升级所要用到的模块和将遇到的问题，而这些都是一个项目经理所应该具备的。作为明星员工，就需要具有宏观、远瞻的视野。

（5）有组织，善于表达沟通

身为一名职场人员，必须有良好的组织悟性，要充分了解组织的真实权力形势。明确上面的职位和自己通过奋斗能够获得的职位是什么，这样才具有针对性。另外，还需要具备良好的沟通能力。身为领导，沟通能力大于专业能力。因为领导的主要任务在于和客户、员工打交道，而不是编写代码。

第 10 章　继承与多态

继承性和多态性是面向对象编程语言的最大特点，作为面向对象编程语言的 C#当然也具有这个特点。在 C#中，一个类可以派生出新的类，并且这个新类会继承基类的成员和特性。在实际项目开发的过程中，可以利用继承开发出具有开放性和可扩充性的程序。多态性是指相同的操作和方法可以被用于多种类型的对象，并且能够获得不同的结果。多态性是通过继承实现的，它增强了程序的灵活性和重用性。通过本章能学到如下知识。

- 继承。
- 隐藏和多态。
- 抽象类和抽象方法。
- 使用 base 关键字。
- 密封类和密封成员。
- 根类 object。
- C#4.0 新特性——dynamic 关键字。
- 职场点拨——兼职。

2007 年 XX 月 XX 日，阴

我最近一直很郁闷，上班一年了，总感觉钱不够花。听说同学小 A 刚完成一个兼职，赚的钱相当于他一个月的工资，我听后很是羡慕，看来我也需要寻找一条兼职的门路了。



一问一答

小菜：我身为全职工作人员，也可以做兼职吗？

Wisdom：在不影响本职工作的前提下当然可以。你可以利用下班后的时间做兼职，不但能带来收入，也能拓展你的人脉关系。

小菜：都是从哪儿获取兼职信息的？

Wisdom：“在我们 IT 行业，特别是程序员，寻找兼职的机会很多，例如有很多专业程序兼职信息网站。在本章的最后，我将介绍做兼职的一些基本常识。”

小菜：“嗯，言归正传，本章的继承与多态有什么用？”

Wisdom：“生活中的你需要做兼职来改善自己的生活。同样 C#程序也需要扩展，在传统汇编语言中只能通过数组来存储复杂的数据类型。但是作为面向对象编程语言的最大特点就是语言的继承性和多态性，这样 C#通过继承和多态即可实现程序的扩展。”

10.1 继承

如果 C#内的一个类直接继承了它基类的成员，那么这个类就被称为这个基类的子类或派生类。派生类能够从其基类继承所有的成员，包括变量和函数结合属性等。基类和子类通过继承形成了一种层次结构，在具体项目应用中可以同时完成不同的功能。

10.1.1 类的层次结构

通过类的继承后，将会在项目中生成一个层次结构，所以在使用继承前，应该确定各类之间的层次关系。例如，有一个 mm 类表示子类，一个 en 类表示派生类，一个 fr 表示派生类。那么就可以根据它们间的对应关系画出层次结构图，如图 10-1 所示。

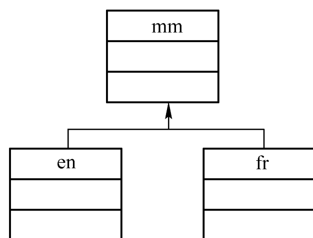


图 10-1 继承关系层次结构图

在图 10-1 所示的层次结构关系中，en 类和 fr 类都是 mm 类的子类，所以 en 和 fr 都继承 mm 的成员。在声明基类时，应该定义其子类中共同拥有的数据成员，例如函数成员和数据成员等。看下面的代码：

```
public class mm{
private string aa;
protected string name;
public string dui;{
get
  {
    return name;
   }
set
  {
    name=value;
}
}
public mm()
  {
}
   public mm(string name)
     {
        this.name=name;
}
   public void Chuli()
     {
        Console.WriteLine("生成球队!!!! ");
}
}
```

上述代码中，在基类 mm 内定义 aa、name 和 dui 三个变量成员。

10.1.2　声明继承

C#在声明类时定义继承，具体使用格式如下：

```
修饰符 class 类名 : 基类
  {
    代码块
}
```

在子类和它的继承基类之间用冒号“:”隔开，例如在下面的代码中，声明了类 nn 的基类是 mm。

```
public class mm{
 protected int aa;
 protected int bb;
 private cc;
public class nn:mm{
 public int Width{
      get {
          return aa;
       }
  }
 public int Height{
     get
      {
        return bb;
      }
  }
public nn(){
  }
public nn(int x,int y){
        aa=x;
        bb=y;
   }
public int mianji(){
        return aa*bb;
    }
 }
}
```

在上述代码中，子类 nn 继承了基类 mm 的成员 aa 和 bb。

10.1.3　继承规则

C#的继承规则如下。

1）除 object 类外，每个类有且只有一个基类。如果在程序中没有显式指定类的基类，那么它的直接基类就是 object。

2）无论基类成员的访问性如何，所有的基类成员都能被其子类所继承，构造函数和析构函数除外。但是需要注意的是，虽然子类能够继承基类的成员，但是不能保证在子类中可以使用这些成员，这取决于成员的可访问性，具体说明如下。

- 在基类内的 public、internal、protected、internal protected 成员被继承后，在子类中仍然是上述访问性成员，并且在子类中可用。
- 在基类内的 private 成员被继承后，在子类中依旧是 private 成员，但是在子类中不可用。

例如在 10.1.2 所示的代码中，nn 类能从基类 mm 继承如下变量成员。

- 变量 aa：类型为 protected，所以在子类 nn 中可用。
- 变量 bb：类型为 protected，所以在子类 nn 中可用。
- 变量 cc：类型为 private，所以在子类 nn 中不可用。

3）子类可以扩展它的直接基类，能够在继承基类的基础上添加新的成员，但是不能删除集成成员的定义。看下面的一段代码：

```
public class mm{
 protected int aa;
 protected int bb;
 private cc;
public class nn:mm{
 public int Width
   {
       get
         {
           return aa;
         }
   }
  public int mianji()
    {
           return aa;
     }
 }
}
```

在上述代码中，虽然子类 nn 继承了基类 mm 的 aa、bb 和 cc 三个成员，但是在子类 nn 中仍然可以声明 int 类型成员 Width、int 类型成员 mianji()。虽然能够在子类中新增成员，但是不能删除继承的 aa、bb 和 cc 这三个成员。

4）继承是可以传递的。例如，如果 C 从 B 派生，而 B 从 A 派生，那么 C 就会继承在 B 中声明的成员，也能继承在 A 中声明的成员。但是各类中使用这些成员时，也需要遵循成员的可访问性。具体说明如下。

- 在类 A 和 B 内的 public、internal、protected、internal protected 成员，在 C 中依旧是上述访问性成员，并且在 C 中可用。
- 在 A 和 B 内的 private 成员被继承后，在 C 中依旧是 private 的成员，但是在 C 中不可用。

5）不能循环继承类。例如，下面的代码是错误的：

```
class a:b{
}
class b:c{
}
class c:a{
}
```

因为上述代码中的类存在循环依赖关系，所以运行后将会出现编译错误。同样，下面的代码也是错误的：

```
class a:b.c{
}
class b:a
{
  class c
   {
   }
}
```

6）类的直接基类必须至少和类的本身具有相同的可访问性。例如下面的代码是错误的：

```
private class mm{
 protected int aa;
 protected int bb;
 private cc;
public class nn:mm{
 public int Width
   {
       get
         {
            return aa;
         }
   }
 }
}
```

因为如果基类的可访问性小于子类的可访问性，就没有继承的必要了。

7）在派生子类中可以声明与继承基类成员相同名称的成员，这样就将基类中的这些成员隐藏了。

8）类可以声明虚方法、虚属性和虚索引器，派生子类可以重写上述虚成员。

9）C#子类只能继承一个直接基类，但是支持基于接口的多重继承。

10.2　C#的隐藏

在 C#子类中，可以使用与基类成员名相同的名称或签名来定义成员，这样在子类中就

不能访问基类的同名成员，这被称为子类成员隐藏了基类的成员。C#中的隐藏主要包括如下三种形式。

1）子类中引入常量、字段、属性、事件等会把基类中的同名成员隐藏起来。

2）子类引入的函数方法隐藏所有同名的非方法基类成员，以及所有具有相同签名的积累方法。

3）子类隐藏的索引器隐藏所有具有相同签名的基类索引器。



因为 C#不能隐藏运算符，所以在派生子类中不能声明与其基类相同的运算符。

看下面的一段代码：

```
class mm
{
   public void chuli()
     {
     }
}
class nn:mm
{
   public void chuli()
      {
   }
}
```

在上述代码中，子类 nn 隐藏了从基类 mm 中继承的成员 chuli()。这样虽然不会有语法错误，但是运行后会导致编译错误。为避免上述错误，可以使用关键字 new 来修饰子类成员。例如下面的代码：

```
class mm{
   public void chuli()
     {
     }
}
class nn:mm{
   new public void chuli()
     {
   }
}
```

在上述代码中，使用了关键字 new 来明确设置要隐藏基类的同名成员 chuli()。这样指定了子类内的 chuli()是新的，并要有意隐藏继承成员。

10.3　C#的多态

多态是一种特性，在具有继承关系的 C#类中，不同对象的签名、相同函数的成员可以

有不同的实现方式，这样就会产生不同的执行结果。C#中所涉及的多态性，大多是指运行时的多态性，即在调用具有相同继承关系的类的相同签名的函数成员时，直至程序运行完毕后才能确定调用哪个类的实例的构造函数成员。C#的运行多态性是通过在子类中重写基类的虚方法，或函数的成员来实现的。

10.3.1　虚方法和虚方法重写

在 C#中，使用关键字 virtual 定义虚方法，没有用 virtual 关键字定义的方法则被称为非虚方法。使用虚方法的具体格式如下：

```
修饰符 virtual 方法()
    {
    }
```



关键字 virtual 不能和 static、abstrct 或 override 修饰符同时使用。并且因为虚拟成员不能是私有的，所以 virtual 也不能和 private 同时使用。

如果在声明某方法中使用 override 修饰符，则称这个方法为重写方法。重写方法用相同的签名来重写所继承的虚方法。其中虚方法的声明用于引入新方法，而重写方法的声明则使从基类继承来的虚方法专用化。



关键字 override 不能和 static、abstrct 或 virtual 修饰符同时使用。并且重写方法只能用于重写基类的虚方法。

非虚方法的实现是不变的，无论是在声明方法类的实例中，还是在派生子类的实例中，当调用这个非虚方法时，它的实现都是相同的。也就是说，派生子类不能改变基类中声明的非虚方法实现。

实例 43：在继承类关系中使用方法

下面通过一个简单实例来说明非虚方法在继承类关系中的使用流程，本实例保存在“光盘：\daima\10”文件夹内，项目名为 noxu。实例的功能是根据定义的方法输出对应处理结果，实现文件的主要代码如下：

```
namespace noxu{
    class class1 {
        public void F()
        {
            Console.WriteLine("class1.F()\n");
        }
    }
    class class2 : class1
    {
        new public void F()
        {
```

```
                Console.WriteLine("class2.F()\n");
            }
        }
        class Test
        {
            static void Main()
            {
                class1 a = new class1();
                class2 b = new class2();
                class1 c = b;
                a.F();
                b.F();
                c.F();
                Console.ReadKey();
            }
        }
    }
```

上述实例代码的设计流程如下。

1）定义基类 class1，并在其中声明方法 F()。

2）定义 class1 的派生子类 class2，并使用 new 关键字将方法 F()隐藏。

3）定义类 Test。

4）分别创建基类 class1 的对象 a，子类 class2 的对象 b。

5）设置对象 c 编译时的类型为 class1，运行时的类型为 class2。

6）分别通过对象 a、b 和 c 调用方法 F()。

执行后将分别调用对应的处理方法并输出对应的文本，如图 10−2 所示。

```
class1.F()

class2.F()

class1.F()
```

图 10−2　实例执行结果

从图 10−2 所示的执行结果可以看出，只要通过声明为 class1 的对象来引用方法 F()，就总是会调用 class1 的 F()。

10.3.2　重写方法的特点

重写方法通常是由 override 重写的那个方法。在 C#中，只有包含 override 关键字的方法才能重写另一个方法。否则，声明一个与从基类继承来的具有相同签名的方法，只会隐藏被继承的基类方法。看下面的一段代码：

```
class mm{
    public virtual void chuli()
    {
    }
}
class nn:mm{
    public virtual void chuli()                  //警告
    {
```

```
        }
    }
```

在上述代码中，因为类 nn 内的方法 chuli()没有包含 override 关键字，所以不能重写 mm 内的方法 chuli()。反之，nn 内的方法 chuli()隐藏了 mm 中的方法，并且因为在声明中没有使用 new 修饰符，所以会发生运行错误警告。看下面的一段代码：

```
class mm{
   public virtual void chuli()
     {
     }
}
class nn:mm{
   new private void chuli()            //隐藏 mm.chuli()
     {
     }
}
class zz:nn{
   public override void chuli()        //重写 mm.chuli()
     {
     }
}
```

在上述代码中，nn 内的方法 chuli()隐藏了从 mm 中继承的方法 chuli()。因为 nn 中的 chuli()有访问权限的问题，它的访问范围只包括在 nn 类内，而在 zz 类内没有权限。所以，允许 zz 中的 chuli()声明重写从 mm 类继承的 chuli()。

在 C#中的虚方法和非虚方法的调用原则如下。

1）当调用虚方法时，对象的运行类型决定了被调用的函数实现。

2）当调用非虚方法时，被调用函数则取决于对该类派生程度最大的实现。

10.3.3 多态的应用

如果在某个基类中使用 virtual 声明了方法 F()，则在子类中使用 override 重写方法 F()。如果使用编译时类型为基类对象 D 来引用方法 F()，则在编译时就不能确定实际调用的方法是基类的 F，还是子类的 F。而只能根据对象 D 在运行时的类型来确定，这就使整个程序有不同的处理结果。这就是 C#的多态性的最大好处。

多态性对具体的编程项目有巨大的作用。例如，可以将一组具有共同基类的对象放到一个特定数组内，然后通过声明来为基类的对象调用它们的同名方法，这样不同类的对象就能分别调用它们各自对应的方法。从而简化了代码编写量，提高了开发效率。

1．处理方法多态性

处理方法多态性即在各类中方法具有的多态性。

实例 44：根据定义的方法执行对应的处理方法

下面将通过一个简单实例来说明使用方法多态性的具体流程，本实例保存在“光盘：\daima\10”文件夹内，项目名为 way。实例的功能是根据定义的方法执行对应的处理方法。

实例文件的主要代码如下：

```
namespace way
{
    public class club
    {
        public virtual void F()
        {
            Console.WriteLine("生成\n");
        }
    }
    public class en : club
    {
        public override void F()
        {
            Console.WriteLine("英国\n");
        }
    }
    public class it : en
    {
        public override void F()
        {
            Console.WriteLine("意大利\n");
        }
    }
    public class sp : club
    {
        public override void F()
        {
            Console.WriteLine("西班牙\n");
        }
    }
    public class ge : club
    {
        public override void F()
        {
            Console.WriteLine("德国\n");
        }
    }
    class Test
    {
        static club[] Read()
        {
            int clubNum = 4;
            club[] clubs = new club[clubNum];
            clubs[0] = new en();
            clubs[1] = new ge();
```

```
            clubs[2] = new it();
            clubs[3] = new sp();
            return clubs;
        }
        static void F(club[] clubs)
        {
            for (int i = 0; i < clubs.Length; ++i)
            {
                clubs[i].F();
            }
        }
        public static void Main()
        {
            club[] clubs = Read();
            F(clubs);
            Console.ReadKey();
        }
    }

}
```

上述实例代码的设计流程如下。

1）定义基类 club，并在其中声明虚方法 F()。

2）定义 club 的派生子类 en，并重写基类 club 的方法 F()，用于输出英国。

3）定义 club 的派生子类 it，并重写基类 club 的方法 F()，用于输出意大利。

4）定义 club 的派生子类 sp，并重写基类 club 的方法 F()，用于输出西班牙。

5）定义 club 的派生子类 ge，并重写基类 club 的方法 F()，用于输出德国。

6）通过 Read()获取类型，并设置球队所属类型种类数为 4。

7）定义方法 F()，根据获取的球队类型调用对应的处理函数。

8）在 Main()中分别调用对应的处理方法。

执行后将分别调用对应的处理方法而输出显示对应的文本，如图 10−3 所示。

```
英国
德国
意大利
西班牙
```

图 10−3　实例执行结果


多学一招

在 C#中多态性的定义是：同一操作作用于不同的类的实例，不同的类将进行不同的解释，最后产生不同的执行结果。C#支持如下两种类型的多态性。

（1）编译时的多态性

编译时的多态性是通过重载来实现的。对于非虚的成员来说，系统在编译时，根据传递的参数、返回的类型等信息决定实现何种操作。

（2）运行时的多态性

运行时的多态性就是指直到系统运行时，才根据实际情况决定实现何种操作。C#中，运行时的多态性通过虚成员实现。

2．更多成员的多态性

除了虚方法和方法重写外，C#还支持属性、索引器和事件的虚重写处理，即通过 virtual 和 override 对属性、索引器和事件进行修饰，实现多态性效果。

10.4 使用 base 关键字

base 关键字和 this 关键字的用法类似，base 关键字有如下两个作用：

1）进行 base 访问。

2）在声明构造函数时指定最先执行的构造函数。

接下来开始讲解 base 关键字上述两个作用的基本知识。

10.4.1 base 访问

使用 base 关键字后，能够使用简写方式在派生类中调用基类成员。上述功能可以通过如下两种方法实现。

1）使用点运算符“.”实现，具体格式如下：

```
base.标识符
```

2）使用中括号对实现，具体格式如下：

```
base[表达式列表]
```

通过 base.可以访问当前类成员所隐藏的基类成员。



base 只能在实例构造函数、实例方法或实例访问器的语句块中使用。当“base.”出现在某个类中时，点运算符“.”后面必须紧跟该类基类的一个成员。

在程序编译时，base.和 base[]形式的表达式分别等价于((基类)this).和 ((基类)this)[]。例如：

```
class mm{
   public void F()
     {
     }
   public int this [int nIndex]
     {
       get{};
     set{};
     }
}
class nn
{
   public chuli()
     {
         int x=base[0];
```

```
                base.F();
            }
        }
```

在上述代码中，“base[0]”等价于“((mm)this)[0]”；“base.F()”等价于“((mm)this).F()”。

当使用 base 引用访问虚函数成员时，通过查找该函数成员相对于此类派生程度的最大的实现程度，来确定调用哪个函数成员的方法。为此在 virtual 函数成员的 override 中，可以使用 base 访问来调用该函数成员的被继承了的实现。看下面的一段代码：

```
        class mm
        {
           int x;
           public virtual void F()
             {
               Console.WriteLine(“x={0}”,x);
             }
        }
        class nn:mm
        {
           int y;
           public override void F()
           {
               base F();
               Console.WriteLine(“y={0}”,y);
           }
        }
```

在上述代码中，重写声明通过 base 来访问已重写了的基方法。

10.4.2　base 构造函数指定

base 可以在声明构造函数时，指定最先执行的构造函数。具体格式如下：

```
        访问修饰符 类名([形参]) : base([实参])
          {
            处理语句块
          }
```

其中，base 所表示的是该类的基类所声明的、形参与实参最为匹配的另一个实例的构造函数，此基类的构造函数会在执行这个当前声明的构造函数之前执行。

实例 45：根据参数的匹配度按顺序执行对应的方法

下面将通过一个实例来说明使用 base 指定构造函数的具体流程。本实例保存在“光盘：\daima\10”文件夹内，项目名为 baseshiyong。实例的功能是根据参数的匹配度按顺序执行对应的方法。实例文件的主要代码如下：

```
        namespace baseshiyong
```

```
    {
         class class1
        {
            public class1(int n)
            {
                Console.WriteLine("class1.class1(int n)\n");
            }
             public class1(string s,int n):this(0)
             {
                 Console.WriteLine("class1.class1(string s,int n)\n");
             }
        }
         class class2 : class1
        {
           public class2(int n):base(0)
            {
                Console.WriteLine("class2.class2(int n)\n");
            }
        }
        class Test
        {
            public static void Main()
            {
                class2 b=new class2(1);
                Console.ReadKey();
            }
        }
    }
```

上述实例代码的设计流程如下。

1）定义基类 class1。

2）在基类 class1 内分别定义两种参数类型的构造函数：class1 (int n)和 class1 (string s,int n)。

3）定义 class1 的派生子类 class2。

4）在类 class2 内 class2 的构造函数 nn(int n)。

5）定义 class2 类型的对象 b。

6）在 Main()中分别调用对应的处理方法。

执行后分别调用对应的处理方法以输出显示对应的文本，如图 10-4 所示。

```
file:///E:/机械工业/c#/daima/10/ba
class1.class1(int n)

class2.class2(int n)
```

图 10-4　实例执行结果

由图 10-6 所示结果可以看出，在执行类 class2 的构造函数 class2 (int n)前，预先执行了参数最为匹配的基类构造函数 class1 (int n)。



多学一招

对于 C#中的构造函数，其调用的具体时机是最为重要的。要想掌握其调用时机，必须

掌握它的如下四个特点。

- 调用基类上已被其他方法重写的方法。
- 指定创建派生类实例时应调用的基类构造函数。
- 基类访问只能在构造函数、实例方法或实例属性访问器中进行。
- 从静态方法中使用 base 关键字是错误的。

概括起来就是“先执行基类的方法再执行派生类的方法”。base 关键字用于从派生类中访问基类成员时使用，即使基类的方法已经在派生类中重写，仍可以使用 base 关键字调用。而且，在创建派生类的实例时，可以使用 base 关键字调用基类的构造函数。使用 base 关键字只能访问基类的构造函数、实例方法和实例属性，不能访问基类的静态方法。

10.5 抽象类和抽象方法

抽象类和抽象方法是 C#中的重要概念之一，通过抽象类和抽象方法可以实现特定的处理功能。本节将分别简要介绍 C#中抽象类和抽象方法的基本知识。

1．抽象类

抽象类不和具体的事物相联系，它只能表达一种抽象的概念，只为它的派生子类提供一个公共接口界面。抽象类是不完整的类，在定义时需要使用 abstract 修饰符。具体格式如下：

```
访问修饰符 abstract class 类名 : 基类\接口
  {
    处理语句块
  }
```

C#抽象类主要有如下四个特点。

1）抽象类不能被直接实例化，并且在使用 new 运算符时会导致编译错误。

2）允许抽象类包含抽象成员。

3）抽象类不能被封装。

4）当从一个抽象类中派生出一个非抽象类时，需要在非抽象子类中重写抽象方法，以便使它独立实现某特定功能。

2．抽象成员

因为不用声明抽象类的具体实现，所以只需要预留一个接口即可作为派生子类的继承界面，这些成员就称为抽象成员。抽象成员可以是方法、属性、索引器和事件。

C#抽象成员的主要特点如下。

1）只能在抽象类中定义抽象成员。

2）在声明类的成员时，使用修饰符 abstract 即可定义一个抽象成员。

3）抽象成员不能是私有的，所以不能和 private 修饰符同时使用。

实例 46：根据定义的方法执行对应的处理方法

下面通过一个简单实例来说明使用抽象类和抽象方法的具体流程，本实例保存在“光盘：\daima\10”文件夹内，项目名为 chouxiang。实例的功能是根据定义的方法执行对应的

处理方法。实例文件的主要代码如下：

```
namespace chouxiang
{
    public abstract class club
    {
        public abstract void F();
    }
    public class qiudui : club
    {
        public override void F()
        {
            Console.WriteLine("生成英国的\n");
        }
    }
    public class en : qiudui
    {
        public override void F()
        {
            Console.WriteLine("生成德国的\n");
        }
    }
    public class it : club
    {
        public override void F()
        {
            Console.WriteLine("生成意大利的\n");
        }
    }
    public class sp : club
    {
        public override void F()
        {
            Console.WriteLine("生成西班牙的\n");
        }
    }
    class Test
    {
        static club[] Read()
        {
            int clubNum = 4;
            club[] clubs = new club[clubNum];
            clubs[0] = new qiudui();
            clubs[1] = new sp();
            clubs[2] = new en();
            clubs[3] = new it();
            return clubs;
```

```
            }
            static void F(club[] clubs)
            {
                for (int i = 0; i < clubs.Length; ++i)
                {
                    clubs[i].F();
                }
            }
            public static void Main()
            {
                club[] clubs = Read();
                F(clubs);
                Console.ReadKey();
            }
        }
    }
```

上述实例代码的设计流程如下。

1）使用 abstract 定义抽象类 club，并在其中声明抽象方法 F()。

2）定义 club 的派生子类 qiudui，并重写其基类 club 的抽象方法 F()。

3）定义 qiudui 的派生子类 en，并重写其基类的方法 F()，用于输出英国球队。

4）定义 club 的派生子类 it，并重写其基类的方法 F()，用于输出意大利球队。

5）定义 club 的派生子类 sp，并重写其基类的方法 F()，用于输出西班牙球队。

6）通过 Read()获取球队类型，并设置球队所属类型种类数为 4。

7）定义方法 F()，根据获取的球队类型调用对应的处理函数。

8）在 Main()中分别调用对应的处理方法。

执行后将分别调用对应的处理方法，输出显示对应的文本，如图 10−5 所示。

生成英国的
生成西班牙的
生成德国的
生成意大利的

图 10−5　实例执行结果


多学一招

抽象类能够被用于类、方法、属性、索引器和事件，使用 abstract 在一个类声明中表示该类倾向。要作为其他类的基类成员被标识成 abstract，或被包含进一个抽象类，必须被其派生类实现。抽象方法有以下特征:

- 一个抽象方法可以看做是一个虚函数。
- 抽象方法的声明只能在抽象类中。
- 因为抽象方法声明只提供一个无实现的方式，没有方法体。
- 抽象方法体的实现被覆写方法提供，覆写方法是一个非抽象类的成员。
- 抽象属性的行为和抽象方法相像，除了不同的声明形式。
- 在一个静态属性中使用 abstract 是一个错误。

10.6　密封类和密封成员

如果所有的类都能被继承，将会导致继承的滥用。会使整个项目程序的结构变得错综复

杂，不利于对系统的维护。因此需要根据具体需要，对类和其成员进行密封。在本节的内容中，将分别对密封类和密封成员的基本知识进行简要说明。

1．密封类

在 C#中，通常将没有必要封装处理被继承的类进行密封，这样被密封的类称为密封类。当一个类被密封后，就不能被其他的类所继承。

在 C#中，使用 sealed 修饰符来密封一个类，具体格式如下：

```
访问修饰符 sealed class 类名 : 基类\接口
    {
        语句块
    }
```

在声明类时，当用 sealed 修饰符修饰后就构成了密封类。如果这个密封类被其他类继承后，就会产生编译错误。



一个密封类不能同时是抽象类，所以 sealed 和 abstract 不能同时使用。例如，下面的一段代码将会发生变异错误：

```
public sealed class nn:mm
      {
        ......
      }
}
public class zz:nn
      {
        ......
      }
}
```

在上述代码中，类 nn 被声明为密封类，所以当类 zz 继承类 nn 时，将会出现错误。

通过密封类可以防止程序内发生意外的派生。因为密封类不能被继承，所以对密封类实例虚函数的调用可以转换为非需调用来处理，这样可以提高程序的运行性能。

2．密封成员

在 C#的非密封类中，如果要防止一个实例成员在子类中被重写，可以将该成员设置为密封成员。具体方法是，在该成员声明时使用修饰符 sealed 来修饰。C#的密封成员可以是密封方法、密封属性、密封索引和密封事件，其中密封方法在现实中最为常见。



1）并不是所有的函数方法都能被密封。密封方法只能用于对基类的虚方法进行重写，并提供具体实现。所以在密封方法声明时，sealed 修饰符需要和 override 修饰符同时使用。通过使用 sealed 修饰后，可以防止派生子类重写该方法。

2）两个 sealed 修饰符不能同时使用。

看下面的一段代码：

```
using System;
class mm
{
    public virtual void F()
      {
        Console.WriteLine(“mm.F()”);
      }
    public virtual void A()
      {
        Console.WriteLine(“mm.A()”);
      }
}
class nn:mm
{
    sealed override public void F()                //密封并重写
      {
        Console.WriteLine(“nn.F()”);
      }
      override public void A()
      {
        Console.WriteLine(“nn.A()”);
      }
}
class zz:nn
{
      override public void F()                     //不能重写密封方法
      {
        Console.WriteLine(“zz.F()”);
      }
      override public void A()
      {
        Console.WriteLine(“zz.A()”);
      }
}
```

在上述代码中，类 nn 提供了如下两个重写方法，具体说明如下。

1）密封方法 F()，在 nn 的子类 zz 中，此方法不能被重写。

2）没有密封方法 A()，在 nn 的子类 zz 中，此方法可以被重写。

10.7　C#静态类

在 C#中，通常将不能实例化且只包含静态成员的类声明为私有构造函数的密封类。因为这种类变量和参数没有意义，所以对这种类进行实例化或派生也是没有实际意义的。为了

防止上述类被实例化或派生，在 C#中引入了静态类这一概念。静态类既不能被实例化，也不能被派生。

在 C#中使用 static 修饰符修饰静态类，具体使用格式如下：

```
访问修饰符 static class 类名
    {
      语句块
    }
```

静态类在使用时必须遵循如下三条规则。

1）静态类不能包含 sealed 和 abstract 修饰符。

2）在声明静态类时，不能显式指定基类或所实现接口的列表。因为静态类都是从 object 隐式继承而来的。

3）静态类不能包含具有可访问性的 protected 或 protected internal 成员。

实例 47：用静态类计算对应图像的面积

下面通过一个简单实例来说明使用静态类的具体方法，本实例保存在“光盘：\daima\10”文件夹内，项目名为 jingtaiclass。实例的功能是根据定义的静态类执行对应图像的面积计算方法。实例文件的主要代码如下：

```
namespace jingtaiclass
{
    public static class mianji
    {
        public const double pai = 3.14;
        public static double yuan(double radius)
        {
            return 3.0 * pai * radius;
        }
        public static double juxing(double width, double height)
        {
            return width * height;
        }
        public static double zheng(double width)
        {
            return width * width;
        }
    }
    class Test
    {
        public static void Main()
        {
            double mianji1 = mianji.yuan(3.77);
            double mianji2 = mianji.juxing(3.8, 9.2);
            double mianji3 = mianji.zheng(4.77);
            Console.WriteLine("圆的面积为：{0}\n", mianji1);
```

```
                Console.WriteLine("矩形的面积为：{0}\n", mianji2);
                Console.WriteLine("正方形的面积为：{0}\n", mianji3);
                Console.ReadKey();
            }
        }

    }
```

上述实例代码的设计流程如下。

1）使用 static 定义静态类 mianji。

2）定义 double 变量 pai，并设置其值为 3.14。

3）定义圆面积的计算方法 yuan()。

4）定义矩形面积的计算方法 juxing()。

5）定义正方形面积的计算方法 zheng()。

6）分别定义类 mianji 的实例对象 mianji1、mianji2 和 mianji3。

7）根据参数选择执行对应的面积计算方法。

执行后将分别调用对应的面积计算方法输出对应图形的面积，如图 10-6 所示。

```
圆的面积为：35.5134
矩形的面积为：34.96
正方形的面积为：22.7529
```

图 10-6　实例执行结果


多学一招

在使用 C#静态类时必须注意如下三点:

1）一个类要想成为静态类，必须使用 static 修饰符显式的进行定义，否则不会自动成为静态的类。

2）只能使用静态类来直接引用它的成员，不能将静态类作为基类、成员构造类型、泛型变量和类型参数约束。

3）静态类不能被用于数组类型、指针类型、new 表达式、强制转换表达式、is 表达式、as 表达式、sizeof 表达式和默认值表达式。

10.8　C#接口

通常将类看成是为实现某项目功能的模板，将接口看成是描述任何类的一组行为。通过接口可以指定项目中各类的运行协议，通过这个协议可以将各个独立的类整合起来，从而使它们共同完成特定的功能。

10.8.1　定义接口

在接口内只包含函数成员的数据结构是引用类型的一种。在 C#中，通过关键字 interface 来声明接口，具体定义格式如下：

```
接口修饰符 interface 接口名 : 基类列表
  {
    处理语句块
```

```
    }
```

C#接口是一种标识符，所以应该遵循标识符的命名规则。例如，下面的一段代码是一个最简单的接口定义形式：

```
interface MyInterface
  {
  }
```

在接口内可以包含一些成员来实现具体的功能，接口内的成员必须遵循如下四点要求。

1）接口内的成员必须是方法、属性、事件和索引器内的一种或几种类型。

2）接口不能包含常量、字段、运算符、实例构造函数、析构函数或类型，也不能包含任何种类的静态成员。

3）接口只包含方法、属性、事件和索引器的签名，而不提供它所定义的成员实现。

4）接口成员都是 public 类型的，但是不能使用 public 来修饰。

例如在下面的代码中，在接口内声明了一个属性和一个方法：

```
interface MyInterface {
     string mm
       {
            get;
            set;
       }
     void chuli();
       {
       }
}
```

从上述代码中可以看出，属性 mm 和方法 chuli()都没有明确使用 public 修饰符，但是实际上它们是默认使用 public 类型的。

接口也可以继承，具体说明如下。

- 接口的继承只说明继承，而不说明具体的实现。
- 接口允许多继承，即一个子接口可以有多个父接口。

一个接口可以从 0 个或多个接口继承，被继承的接口称为该接口的显式基接口。当一个接口具有多个显式基接口时，在声明中要用逗号“,”来分隔开。例如，下面代码中的接口 mm 有 nn 和 zz 两个显式基接口。

```
interface mm : nn,zz
  {
     string mm
      {
           get;
           set;
      }
  }
```

在现实应用中，可以通过接口的继承特性来实现接口组合机制，这样可以用很少的代码实现复杂的功能。而且还能和类实现的模板相结合，设计出面向对象编程的项目程序。例如下面的代码：

```
interface mm{
      object A();
   }
interface nn{
      void B();
   }
interface zz
   {
      void C();
   }
interface tt
   {
      void D();
   }
interface ff:mm,nn,zz,tt
   {
   }
```

这样，接口 ff 将组合 mm、nn、zz 和 tt，同时可以使用方法 A、B、C 和 D。

10.8.2　接口的实现和继承

接口的实现是指在某个类中继承某个接口时的过程。虽然一个类只能继承一个直接接口，但是它可以实现任意数量的接口。所以接口的实现具有多继承性的特性，为此，在声明类时应该在基类列表中包含类所实现的接口名称。

例如在下面的代码中，类 mm 实现了接口 interface1 和 interface2。

```
interface interface1
   {
     object A();
   }
interface interface2
   {
     int void B();
   }
class mm: interface1, interface2
   {
    public object A();
      {
      }
    public int void B();
      {
```

```
            }
        }
```

如果在某个类中实现了一个接口，那么它也隐式实现了该接口的所有基接口。虽然在类的基类列表中没有显式列出所有基接口，但是隐式实现了该接口的所有基接口。看下面一段代码：

```
        interface mm
          {
             string neme
              {
                    get;
                    set;
              }
            void chuli();
          }
        interface nn:mm
          {
             void aa();
          }
        class lei : nn
          {
              get
                {
                }
              set
                {
                }
           }
               public void chuli();
                {
                }
               public void aa();
                {
                }
           }
```

在上述代码中，虽然类 lei 内的基类列表中没有接口 mm，但是同时实现了 mm 和 nn。

类会继承由其基类提供的所有接口的实现，如果不显式的重新实现接口，派生类将无法以任何方式更改它从基类继承来的接口映射。

10.8.3　显式接口的成员

当在 C#中实现接口时，可以在类中声明一个显式接口成员实现。显式接口成员实现是指在声明方法、属性和索引器时，使用完全限定接口的成员名称作为标识符。以显式接口成员实现方式生成的成员叫做显式成员。

接口成员完全限定名的使用格式如下：

```
声明的该成员接口名.成员名
```

要实现一个显式接口成员，必须同时具备如下两个条件。

1）在声明类时，必须在其基类列表中指定一个接口，并且这个接口必须包含一个与该显式接口成员的完全限定名、类型和参数类型完全相同的成员。

2）接口成员的完全限定名必须引用声明该成员的接口名称。

另外，在实现 C#接口成员的完全限定名时，不能包含 abstract、virtual、override、static 等访问修饰符。例如，在下面的代码中分别使用了显式接口成员实现：

```
interface mm
   {
      object chuli();
   }
interface nn
   {
      int AA()
   }
class lei:mm,nn
   {
      object mm.chuli()                          //显式接口成员实现
      {
         Console.WriteLine("调用的是");
      }
 int nn.AA()                                     //显式接口成员实现
      {
         Console.WriteLine("调用的是");
      }
   }
```

显式接口成员属于接口成员，所以不能使用类实例来访问，而只能使用接口实例来访问。并且在使用接口实例来访问时，只能使用该接口成员的名称来引用。显式接口成员可以使用 this 进行访问，也可以直接引用类的成员。但是在类内不能直接引用显式接口成员实现，而需要使用 this 进行显式转换成接口实例后再进行引用。

实例 48：实现显式接口成员

下面将通过实例来说明实现显式接口成员的具体流程。本实例保存在“光盘：\daima\10”文件夹内，项目名为 xianshijiekou。实例的功能是根据定义的接口，在类内执行对应的处理方法。实例文件的主要代码如下：

```
namespace xianshijiekou {
   interface mm
   {
      void Save();
   }
```

```
    interface nn
    {
        void Save();
    }
    class aa : mm, nn
    {
        void mm.Save()
        {
            Console.WriteLine("调用了 mm.Save()\n");
        }
        void nn.Save()
        {
            Console.WriteLine("调用了 nn.Save()\n");
        }
        public void Save()
        {
            Console.WriteLine("调用了 aa.Save()\n");
        }
    }
    class Test
    {
        public static void Main()
        {
            aa doc = new aa();
            mm db = doc;
            nn file = doc;
            doc.Save();
            db.Save();
            file.Save();
            Console.ReadKey();
        }
    }
}
```

上述实例代码的设计流程如下。

1）分别定义接口 mm 和 nn，并在其中分别定义处理方法 Save()。

2）定义类 aa，设置其基接口为 mm 和 nn。

3）在类 aa 内分别使用显式接口成员实现 mm.Save 和 nn.Save。

4）在类 aa 内声明处理方法 Save()。

5）定义类 aa 的实例对象 doc，并调用它的处理方法 Save()。

6）分别定义接口实例对象 db 和 file，并调用它们各自的处理方法 Save()。

执行后将分别调用对应的处理方法输出对应的方法类型，如图 10-7 所示。

```
file:///E:/机械工业/c#/daima/10/xi
调用了aa.Save()

调用了mm.Save()

调用了nn.Save()
```

图 10-7　实例执行结果

从图 10-7 所示的执行结果可以看出，使用显式接口成员实现可以使类 aa 内的三个处理方法 Save()独立运行，各自完成它们的功能，而不会造成冲突。


多学一招

在使用显示接口时必须注意如下两点。

1）显式接口成员属于接口成员，所以不能使用类实例来访问，而只能使用接口实例来访问。并且在使用接口实例访问时，只能使用该接口成员的名称来引用。

2）显式接口成员可以使用 this 进行访问，也可以直接引用类的成员。但是在类内不能直接引用显式接口成员实现，而需要使用 this 进行显式转换成接口实例后再进行引用。

10.8.4　访问接口成员

访问接口成员的方式和访问类成员的方式类似，有如下两种方式：

1）使用点运算符“.”

使用点运算符的具体格式如下：

```
接口类型.接口成员
```

2）使用“[]”对

使用“[]”对的具体格式如下：

```
接口类型[索引器参数列表]
```

1. 多接口处理

如果只是单一继承接口，则成员的查找、方法的调用和索引器的访问规则和类的完全相同。如果是多重继承，为解决多个无关基接口中同名或同签名的接口成员的混乱问题，可以使用显式强制转换来解决。看下面的一段代码：

```
interface mm
   {
        int num
              {
                get;
                set;
              }
   }
interface nn
   {
       void num(int i)
   }
interface zz:mm,nn
   {
   }
class lei
   {
```

```
            void chuli(zz x)
            {
                x.num(1);                          //出现错误
                x.num=1;                           //出现错误
            }
        }
```

在上述代码中，接口 zz 多重继承了接口 mm 和 nn。当在 lei 类中使用“x.num(1)”和“x.num=1”调用方法 num()时，因为不能确定究竟调用的是 mm 的方法还是 nn 的方法，所以会出现运行错误。解决上述错误的方法是，使用显式强制转换，对上述类 lei 的代码进行如下修改：

```
    class lei
        {
            void chuli(zz x)
            {
                ((mm)x).num(1);                    //调用的是 mm 的方法
                ((nn)x).num=1;                     //调用的是 nn 的方法
            }
        }
```

2. 重载类型决策

在选择调用处理方法时，会遵循参加重载类型决策原则。看下面的一段代码：

```
    interface mm {
            void chuli(int i)
        }
    interface nn{
            void num(double i)
        }
    interface zz:mm,nn {
        }
    class lei {
            void text(zz x)
            {
                x.num(1);                          //调用的是 mm 的方法
                x.num(1.1);                        //调用的是 nn 的方法
                ((mm)x).num(1);                    //调用的是 mm 的方法
                ((nn)x).num=1;                     //调用的是 nn 的方法

            }
        }
```

在上述代码中，虽然“x.num(1)”和“x.num(1.1)”没有经过显式强制转换，但是两者的参数类型是不同的，一个是 int 类型的 1，一个是 double 类型的 1.1。所以上述代码运行后，会自动调用对应的处理方法。

3. 直观隐藏

直观隐藏是指在多重继承接口中，如果某成员在任意一个路径中被隐藏，那么它会在所有的访问路径中被隐藏。例如，有 4 个名为 mm、nn、zz 和 tt 的接口。其中 mm 是 nn 和 zz 的基接口，而 nn 和 zz 是 tt 的基接口。如果重新定义接口 nn 的基接口 mm 内的方法 chuli()，那么被隐藏后的方法 chuli()在其他接口内也是被隐藏的。上述接口的关系如图 10-8 所示。

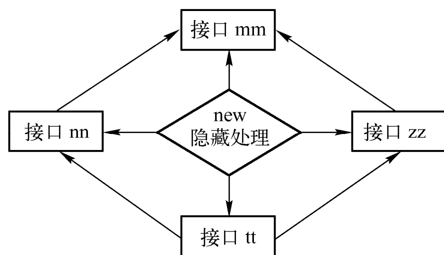


图 10-8　接口关系结构图

上述结构关系的对应代码如下：

```
interface mm
   {
     void chuli(int i)
   }
interface nn.mm
   {
      new void num(int i)
   }
interface zz:mm
   {
      void F()
   }
interface tt:nn,zz
   {
   }
class lei
   {
      void text(tt x)
      {
         x.num(1);                                   //调用的是 nn 的方法
         ((mm)x).num(1);                             //调用的是 mm 的方法
         ((nn)x). num(1);                            //调用的是 nn 的方法
         ((zz)x). num(1);                            //调用的是 mm 的方法
      }
   }
```

10.9　C#根类 object

C#内所有的类都是从根类 object 中继承而来的，类 object 实际上是类 System.object 的缩写。System.object 不仅是 C#类型的根，而且也是.NET 类库的类型根。所以，在.NET 框架内的一切类型，都是直接或间接的从 System.object 中继承过来的。

10.9.1 object 的方法

类 System.object 提供了一系列的虚方法，所有的类都可以直接对它们进行重写。System.Object 方法的具体说明如表 10-1 所示。

表 10-1 System.Object 方法列表

方法	描述
Equals()	验证对象是否等价
GetType()	返回 Type 对象的类型
GetHashCode()	当对象用于集合时，使对象能够提供自己的散列函数
ToString()	提供表示对象的字符串
Finalize()	由析构函数实现，用于清理非内存资源
MemberwiseClone()	创建对象副本
ReferenceEquals()	验证两对象是否引用同一个实例

当 System.object 默认实现时，Equals()和 ReferenceEquals()是基本相同的，唯一的区别是 Equals()可以适用于任何类型，而 ReferenceEquals()只适用于引用类型。

10.9.2 object 的方法应用

当使用值类型对象进行处理时，如果两者相等则 Equals()方法返回 true，否则返回 false。如果处理的是引用类型对象，引用的对象相同则返回 true，否则返回 false。

1．虚方法重写

方法 Equals()既可以是静态方法，也可以是虚方法，通过自定义类可以重写虚方法形式的 Equals()。而方法 ReferenceEquals()只有静态形式。

实例 49：根据创建的对象输出显示两对象的比较结果

下面通将过一个实例来说明使用 Equals()和 ReferenceEquals()的具体流程。本实例保存在“光盘：\daima\10”文件夹内，项目名为 xuwaychong。实例的功能是根据创建的对象输出显示两对象的比较结果。实例文件的主要代码如下所示：

```
namespace xuwaychong {
    class mm
    {
        string name;
        public mm(string name)
        {
            this.name = name;
        }
    }
    class Test
    {
        static void Main()
        {
```

```
            mm mm1 = new mm("张三");
            mm mm2 = new mm("张三");
            Console.WriteLine("两个不同的对象：");
            bool Equals = Object.Equals(mm1, mm2);
            bool ReferenceEquals = Object.ReferenceEquals(mm1, mm2);
            Console.WriteLine("Equals(mm1, mm2)为{0}", Equals);
            Console.WriteLine("ReferenceEquals(mm1, mm2)为{0}", Reference
Equals);
            mm2 = mm1;
            Console.WriteLine("\n引用同一个对象后：");
            Equals = Object.Equals(mm1, mm2);
            ReferenceEquals = Object.ReferenceEquals(mm1, mm2);
            Console.WriteLine("Equals(mm1, mm2)为{0}", Equals);
            Console.WriteLine("ReferenceEquals(mm1, mm2)为{0}", Reference
Equals);
            Console.ReadKey();
        }
    }
}
```

上述实例代码的设计流程如下。

1）定义类 mm，并在其中分别定义字段 name 和参数为 name 的方法。

2）分别定义类 mm 的实例对象 mm1 和 mm2。

3）分别为实例对象 mm1 和 mm2 设置参数为“张三”。

4）分别调用 Equals()和 ReferenceEquals()进行比较处理。

5）通过 WriteLine()输出比较处理的结果。

6）赋值对象 mm1 等于 mm2。

7）分别调用 Equals()和 ReferenceEquals()进行赋值后的比较处理。

8）通过 WriteLine 输出赋值后的比较处理的结果。

上述实例代码执行后，将分别调用比较方法输出对应的结果，如图 10-9 所示。

```
file:///E:/机械工业/c#/daima/10/xuwaychong/bin/Debug/xuchong.EXE
两个不同的对象：
Equals(mm1, mm2)为False
ReferenceEquals(mm1, mm2)为False

引用同一个对象后：
Equals(mm1, mm2)为True
ReferenceEquals(mm1, mm2)为True
```

图 10-9　实例执行结果

从图 10-9 的所示结果可以看出，虽然类 mm 的对象 mm1 和 mm2 的参数都是“张三”，但是两者依旧不是同一个对象。只有使用“mm1=mm2”进行赋值处理后，两者才完全相同。

2．方法重写

当自定义一个类后，即使没有指定基类也会隐式的把 System.object 作为基类。所以在具

体声明时，并不需要明确地将 System.object 指定为基类。在自定义的类中，可以对 System.object 内的所有方法重写，Finalize()和 MemberwiseClone()除外。



1）如果要实现 Equals()或 GetHashCode()的重写，必须对这两个方法全部重写，否则在编译时会出现错误信息提示。

2）GetHashCode()产生的散列值不是唯一的。

3）在 GetHashCode()种不能出现异常。

10.9.3　Finalize()特殊应用

System.object 中的方法 Finalize()比较特殊，不允许使用 C#程序对其进行重写处理或直接调用。但是在 C#程序中，每个类的析构函数都是通过重写 Finalize()实现的。看下面的一段代码：

```
namespace Finalizechuli
{
    class mm
    {
        ~mm()
        {
            Console.WriteLine("重写 mm.");
            Console.ReadKey();
        }
    }
    class Test : mm
    {
        public static void Main()
        {
            mm a = new mm();
        }
    }
}
```

在上述代码中，通过“~”修饰符定义析构函数 mm。当程序运行后将首先执行这个析构函数，然后再运行 Test 内的代码。具体运行结果如图 10-10 所示。

重写 mm.

图 10-10　实例执行结果

稍等几秒后将会运行 Test 内的代码块，控制台自动关闭。



除了前面介绍的几种方法外，表 10-1 所示的 System.object 其他方法的应用比较简单，都和自定义的方法的使用没有特殊的区别。其中的方法 GetType()的功能是返回 Type 对象的类型，这和 typeof 有很大的区别。typeof 运算符针对的是类型名，而 GetType()针对的是对象。

10.10　C#类转换

在 C#程序中，可以在类的对象之间进行引用转换，具体包括隐式转换和显式转换两种。并且上述转换虽然能改变引用的类型，但是不会更改所引用对象的类型和值。

10.10.1　隐式转换

通过类引用的隐式转换，能够实现引用类型之间的转换。因为这种转换都能一次性成功，所以不需要在运行时进行任何检查。C#类的隐式转换主要适用于如下五种情况。

1）从任何引用类型到 object 的转换。

2）从任何派生子类到到其本身基类的转换。

3）如果某类实现了某个接口，则可以实现从这个类到这个接口的转换。

4）从任何派生接口到其本身基接口的转换。

5）从 null 到任何引用类型。

看下面的一段代码：

```
interface mm
   {
     void chuli();
   }
class aa : mm
   {
      public void chuli()
        {
        }
   }
class bb:aa,mm
   {
   }
class lei
   {
      public static void Main ()
      {
         bb cc =new bb();
         aa dd=cc;                                //隐式转换
         mm ee=cc;                                //隐式转换
         obj obj=cc;                              //隐式转换
      }
   }
```

在上述代码中，接口是 mm，类是 aa 和 bb。根据它们间的继承关系和实现关系，在类 lei 内实现了隐式转换处理。具体说明如下。

- “aa dd=cc”，子类到基类的转换。
- “mm ee=cc”，实现类到接口的转换。

- “obj obj=cc”，类到 object 的转换。

10.10.2 显式转换

使用类引用的显式转换，能够在运行时实现需要的检查处理，以确保引用类型之间的转换是正确的。因为这种转换不能确保完全成功，所以需要在运行时进行检查。C#类的显式转换主要适用于如下五种情况。

1）从 object 到任何其他引用类。

2）从任何基类到到其本身派生子类的转换。

3）从任何类到任何接口的转换，条件是这个类是未密封的并且它不能实现这个接口。

4）从任何接口到到任何接口的转换，条件是被转换接口不是目标接口的派生子类

5）从任何接口到任何类的转换，条件是类是未密封的或这个类可以实现这个接口。

为了使显式引用转换能够运行成功，源操作数的值必须是 null 类型的，或源操作数所引用的对象的实际类型必须是一个可通过隐式引用转换为目标类型的类型。如果显式引用转换处理失败，则会引发运行异常。看下面的一段代码：

```
        interface mm
           {
              void chuli();
           }
        interface nn
           {
             void F();
           }
        class aa : nn
           {
               public void F()
                 {
                 }
           }
        class bb:aa,nn
           {
           }
        class lei
           {
               public static void Main ()
               {
                   aa doc =new aa();
                   nn fi=doc;                                //隐式转换
                   object obj=doc;                           //隐式转换
                   mm db=(mm)doc;                            //显式转换
                   bb bb1=(bb)doc;                           //显式转换
                   bb1=(bb)obj;                              //显式转换
                   bb1=(bb)fi;                               //显式转换
                   fi=(nn)db;                                //显式转换
```

```
        }
    }
```

在上述代码中，定义了接口 mm 和 nn，并定义了类 aa 和 bb。根据它们间的继承关系和实现关系，在类 lei 内实现了隐式转换处理和显式转换处理。当运行上述程序后，会显示异常错误。错误发生在“mm db=(mm)doc”这一行，即将类 aa 类型显式转换为接口 mm 的类型。

10.10.3　使用 is 运算符

is 运算符的功能是动态检查程序对象运行时的类型是否和给定的类型相兼容。is 运算符的运算结果是布尔值，具体运算格式如下：

```
表达式（B）  is  类型（T）
```

is 运算符的运算规则如下。

1）如果 B 的编译类型和 T 相同，且存在从 B 编译时的类型到 T 的隐式引用转换，或者装箱转换，具体规则如下。

- 如果 B 是引用类型，则运算结果和 B!＝null 等效。
- 如果 B 是值类型，则运算结果为 true。

2）如果 B 的编译类型和 T 不相同，且存在从 B 编译时的类型到 T 的显式引用转换，或者拆箱转换，具体规则如下。

- 如果 B 的值是 null，则运算结果是 false。
- 如果 B 的值不是 null，假设 R 是 B 引用实例的运行类型，若 R 和 T 的类型相同，或 R 为引用类型并且存在从 R 到 T 的隐式引用转换，或 R 是值类型并且 T 是 R 实现了的接口类型，则运算结果是 true；否则运算结果是 false。

如果不是上述两种情况，则不会实现从 B 到 T 的引用转换或装箱转换，运算结果是 false。

实例 50：使用 is 运算符

下面通将过一个实例来说明使用 is 运算符的具体流程。本实例保存在“光盘：\daima\10”文件夹内，项目名为 ismath。功能是将不同转换表达式的结果输出。实例文件的主要代码如下：

```
namespace ismath
{
    interface mm
    {
        void Save();
    }
    class aa : mm
    {
        public void Save()
        {
        }
```

```
        }
        class bb : aa, mm
        {
        }
        class Test
        {
            public static void Main()
            {
                bb ff = new bb();
                aa zz = new aa();
                mm file = zz;             // 隐式转换：bb 到实现了的接口 mm
                object obj = zz;          // 隐式转换：类 bb 到 object
                Console.WriteLine("表达式：bb is bb 结果{0}\n", ff is bb);
                Console.WriteLine("表达式：zz is bb 结果：{0}\n", zz is bb);
                Console.WriteLine("表达式：file is bb 结果：{0}\n", file is bb);
                Console.WriteLine("表达式：obj is bb 结果：{0}\n", obj is bb);
                Console.WriteLine("表达式：zz is aa 结果：{0}\n", zz is aa);
                Console.WriteLine("表达式：file is aa 结果：{0}\n", file is aa);
                Console.WriteLine("表达式：obj is aa 结果：{0}\n", obj is aa);
                Console.ReadKey();
            }
        }
    }
```

上述实例代码的设计流程如下。

1）定义接口 mm，并在其中定义方法 Save()。

2）分别定义类 aa 和 bb，设置 aa 的基接口是 mm，bb 的基类是 aa，bb 的基接口是 mm。

3）在类 aa 内定义方法 Save()。

4）分别定义类 aa 和 bb 的实例对象 zz 和 ff。

5）分别定义接口 mm 的对象 file，根类 object 的对象 obj。

6）通过 WriteLine()输出各表达式，并输出对应的运算结果。

执行后将分别输出显示各表达式的对应运算结果，如图 10-11 所示。

```
表达式：ff is bb 结果True

表达式：zz is bb 结果：False

表达式：file is bb 结果：False

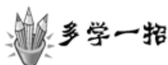
表达式：obj is bb 结果：False

表达式：zz is aa 结果：True

表达式：file is aa 结果：True

表达式：obj is aa 结果：True
```

图 10-11　实例执行结果



is 运算符只能进行引用转换、装箱转换和拆箱转换，而不能进行其他类型的转换处理。C#中的 is 运算符，最重要的应用就是进行强制转换。

因为对象是多态的，所以基类类型的变量可以保存派生类型。若要访问派生类型的方法，需要将值强制转换回该派生类型。不过，在这些情况下，如果只尝试进行简单的强制转换，会导致引发 InvalidCastException 的风险。这就是 C#提供 is 和 as 运算符的原因。可以使用这两个运算符来测试强制转换是否会成功，而没有引发异常的风险。通常，as 运算符更高效一些，因为如果可以成功进行强制转换，它会实际返回强制转换值。而 is 运算符只返回一个布尔值。因此，如果只想确定对象的类型，而无需对它进行实际的强制转换，则可以使用 is 运算符。

10.10.4　使用 as 运算符

运算符 as 的功能是，将某个对象显式的转换为一个指定的引用类型。as 运算符从来不会发生运行异常，这和显式转换有很大的区别。使用 as 运算符的具体格式如下：

```
表达式（B） as 引用类型（T）
```

as 运算符的运算规则如下。

1）如果 B 的编译类型和 T 相同，则运算结果是 B 的值。

2）如果 B 的编译类型和 T 不相同，且存在从 B 编译时的类型到 T 的隐式引用转换，或者装箱转换，则执行这个转换处理，并且该转换的结果就是运算结果。

3）如果 B 的编译类型和 T 不相同，且存在从 B 编译时的类型到 T 的显式引用转换，则执行动态类型检查。具体规则如下。

- 如果 B 的值是 null，则运算结果是具有编译时类型 T 的值 null。
- 如果 B 的值不是 null，假设 R 是 B 引用实例的运行类型：如果 R 和 T 的类型相同，或 R 为引用类型并且存在从 R 到 T 的隐式引用转换，或 R 是值类型并且 T 是 R 实现了的接口类型，则运算结果是由 B 给定的具有编译时类型 T 的引用。

如果不是上述情况，则不会实现从 B 到 T 的引用转换或装箱转换，并且发生编译错误。

看下面的一段代码：

```
interface mm
   {
    void chuli();
   }
interface nn
   {
    void F();
   }
class aa : nn
   {
      public void F()
```

```
            {
            }
        }
    class bb:aa,nn
        {
        }
    class lei
        {
            public static void Main ()
            {
                aa doc =new aa();
                nn fi=doc;                          //隐式转换
                object obj=doc;                     //隐式转换
                //开始 as 运算
                mm db=doc as mm;
                bb bb1= doc as bb;
                bb1= obj as bb;
                bb1= fi as bb;
                fi= db as nn;
                Console.WriteLine("转换完成");
                Console.ReadKey();
            }
        }
```

上述代码运行后，不会引发异常错误，具体如图 10-12 所示。

但是转换失败的对象将为 null，使用这些对象后就会引发使用空引用的异常。

图 10-12　实例执行结果



as 运算符只能进行引用转换和装箱转换，而不能进行其他类型的转换处理。

10.11　C# 4.0 新特性——dynamic 关键字

在 C#新增的功能中，其中一个很重要的就是对动态语言的支持，在.Net framework 4.0 里 C#可以和一些建立在 CLR 基础上的动态语言（譬如 Iron Python）进行很方便的互操作。在语言层面，C#引进了新的关键字 dynamic。

dynamic 关键字用于声明一个动态对象，然后通过该动态对象去调用方法或读写属性。看下面一段代码：

```
    class Program {
    static void Main(string[] args)
    {
    dynamic myObject = GetSomeList();
    myObject.NonExistsMethod();
```

```
}
private static dynamic GetSomeList()
{
return new List<int>();
}
}
```

可以看到方法 GetSomeList()定义的返回类型是 dynamic，上面例子里实际返回的类型是 List。而事实上这里可以返回任何类型，无论 int，string，StringBuilder 都可以，因为已经定义了它的返回类型是“动态”的。

而 myObject 调用了它的方法 NonExistsMethod，正如其名，这个方法在任何地方都是不存在的。但是上述程序可以在 Visual Studio 2010 里编译通过，没有任何错误。如果把 NonExistsMethod 替换成实际存在的方法，例如 Sort()，运行也是没有问题的。也就是说，如果某个变量是 dynamic 类型的，那么编译器允许调用任何存在或者不存在的方法。这些方法会一直到运行时才再被绑定，如果不存在，当然就会有异常抛出。

由此可以看出，dynamic 和 object 很像。一个 dynamic 类型的引用和 object 一样，可以指向任何类型。C#中，方法 GetSomeList()就是这样的用法。但是两者又有点不同，例如可以调用任何方法，这样就增强了动态性。编译上述例子并在 reflector 里查看过后，会发现在 C#编译上确实把它作为了 object，GetSomeList() 会被编译成：

```
[return: Dynamic]
private static object GetSomeList() {
return new List<int>();
}
```

而对 dynamic 对象的调用会被编译成：

```
CallSite<Action<CallSite, object>> site;
site = CallSite<Action<CallSite, object>>.Create(
Binder.InvokeMember(CSharpBinderFlags.ResultDiscarded,
"NonExistsMethod",
null,
typeof(Program),
new CSharpArgumentInfo[] { CSharpArgumentInfo.Create(CSharpArgumentInfo
Flags.None, null) }
));
site.Target.Invoke(site, myObject);
```

由此可见，在 dynamic 背后，NonExistsMethod 被处理成了动态绑定用的字符串。而一个新的类型 System.Runtime.CompilerServices.CallSite 被引入到.Net 4.0 里用来支持这样的动态绑定。

引入 dynamic 的意义如下。

1）可以和动态语言互操作，可以同时享受动态语言的灵活和.Net 的强大

2）可以支持自顶而下的开发方式，从顶层的架构开始搭起会很容易，调用不存在的方

法（甚至这个类都根本不存在）编译也没有问题。



dynamic 的引入可以使 C#和动态语言一样，失去编译器的保护。尤其在对代码重构的时候，如果代码中存在对 dynamic 的滥用，将会造成巨大的麻烦。所以这件武器虽然强大，但同时也是一把双刃剑。

10.12 疑难问题解析

本章详细介绍了 C#语言的继承与多态的基本知识。本节将对本章中比较难以理解的问题进行讲解。

读者疑问：请问，在使用 C#隐藏时应该注意些什么？

解答：在使用 C#隐藏时，应该充分注意如下两点。

1）在子类中隐藏基类成员后并不会删除了基类成员，只是使基类成员在子类中不可见。

2）当某子类 nn 中声明的一个新成员隐藏了某基类 mm 的成员后，只能在新成员的范围内隐藏囊基类 mm 的这个成员。所以，如果这个新成员是 private 类型，则在从子类 nn 派生的子类 zz 中，zz 将会继承 mm 的成员，而不是 nn 的成员。

看下面的一段代码：

```
class mm{
    public static void chuli()
      {
      }
}
class nn:mm{
    new private static void chuli()      //仅在nn内隐藏mm.chuli()
      {
      }
    public static void aa()
      {
      chuli();                           //调用nn.chuli()
      }
}
class zz:nn
    public static void bb(){
      chuli();                           //调用mm.chuli()
      }
}
```

在上述代码中，使用 new 关键字后，仅仅是在类 nn 内隐藏了基类成员 chuli()。所以当在类 nn 内调用 chuli()时，调用的是 nn 的 chuli()；而在类 zz 内调用 chuli()时，调用的是 mm 的 chuli()，因为 nn 中新定义 chuli()是 private 类型的，它在 zz 内是不可访问的。

读者疑问：请问，在使用 C#静态类时应该注意些什么？

解答：在使用 C#静态类时，应该注意如下三点。

1）一个类要想成为静态类，必须使用 static 修饰符显式的进行定义，否则不会自动成为静态的类。

2）只能使用静态类来直接引用它的成员，不能将静态类作为基类、成员构造类型、泛型变量和类型参数约束。

3）静态类不能被用于数组类型、指针类型、new 表达式、强制转换表达式、is 表达式、as 表达式、sizeof 表达式和默认值表达式。



职场点拨——兼职

当今生活中大家遇到了一个很严峻的问题：无论是买房、买车、旅行散心，还是和三五知己结伴聚餐，都得需要钱，但是忽然间发现我们的口袋越来越薄。程序员也是凡夫俗子，也需要赚钱来养家糊口。固定的那些工资有时满足不了我们的消费需求，怎么办？寻找兼职是一个很好解决上述问题的途径。现在 IT 行业外包盛行，很适合发展兼职。程序员们可以通过以下渠道获取兼职。

1）客户。身为开发人员，肯定在做项目的时候接触过客户。曾经为这个客户服务过，再和客户的沟通下，共同完成了客户需要的项目。同时他也认同了自己的技术没问题，之后他就是你的老客户了。经过第一个项目的相互了解，如果再有新的项目时他很可能会继续找自己来完成。

优点是：因为以前曾经合作过，所以在流程上会很顺利。

缺点是：他认识自己的同事或上司，对本职会有影响。

2）朋友介绍。都说朋友多了路好走，日常生活中，朋友和同学经常会介绍来客户。

优点是：对方肯定真实可靠。

缺点是：可能碍于面子，在收入可能会有影响，在日后服务上可能会耗费时间。

3）网络资源。当前互联网迅猛发展，各个兼职类、外包类站点如雨后春笋般纷纷建立。常见的兼职信息网有例威客、365huo 和 CSDN 等。

优点是：信息量大；

缺点是：竞争大，信用度较低。

4）开一个工作室。工作之余，可以叫上三五知己或同学，合伙开一个工作室。这样能有固定的人员，固定的办公地点，更能吸引客户的到来。

但是即使找到了客户，一定能顺利的合作成功吗？在此笔者有以下几个建议供读者参考。

1）首先选择朋友或熟人推荐的，这样双方都比较放心，项目也好拿一些，一般也不会欠款。

2）尽量实现书面化，例如在接活前一定要先让客户把需求写成书面形式，然后根据文本里要求的功能估价。这样对双方都有好处，能够更好的明确自己的职责。

3）一定要明确后续职责，作为程序开发项目，肯定会涉及维护和升级的问题，在开始之前一定要商定好维护和升级的报酬。

4）在实现 Web 项目时，程序和页面一定要分开。这样做的好处是不但分工明确，而且还能提高工作效率，后期美工改起来也不会影响到程序。

5）在开工前一定要清楚客户说的报酬，是税前的还是税后的。

6）申请一个属于自己的服务器上空间，将以前做过的案子都放在网上，特别是 Web 项目。这不但可以展示以前的作品，而且可以展示当前项目的进度。

第 11 章　字符串与正则表达式

字符串是 C#中重要的数据类型之一，通过对字符串的赋值可以实现所需要的功能。正则表达式是一种描述字符串匹配的模式，它能够对字符串实现指定功能的操作。在本章内容中，将简要介绍 C#中字符串与正则表达式的基本知识，并通过具体的实例来介绍其具体使用流程。通过本章能学到如下知识。

- C#字符串。
- 正则表达式。
- 命名可选参数与 COM 互操作。
- 职场点拨——部门间的沟通。

2007 年 X 月 XX 日，晴

我一大早来到公司，经理 A 就把我叫进了办公室说："NB 系统测试后，发现系统和数据之间存在问题，你检查一下。"我检查发现是因为系统升级后，没有进行对应的迁移。下午客户看后说："系统很不符合我们的要求，界面风格和颜色严重违背了我们公司的主题风格，我已经给你们详细的资料和要求文档了，怎么能开发出这个效果呢？"我和其他开发部的同事们面面相觑，都心中暗想："我们怎么没见到详细的需求资料和要求文档呢？产品部只给了两页笼统的说明。"



一问一答

小菜：现在的公司怎么能这样呢，设立一个个不同的部门，沟通起来真够麻烦！

Wisdom：呵呵，不要抱怨了。不同的部门干不同的工作，这样才能提高工作效率，也能发挥不同部门和不同员工的优势。例如你所在的开发部，肯定是开发技术过硬的人员，而产品部，则在客户沟通和产品规划方面占优势！在本章的最后，我将给你指点部门沟通的一些经验！

小菜：嗯，言归正传，本章的字符串与正则表达式，两者之间有什么关系吗？

Wisdom：字符串和正则表达式是一对搭档，正则表达式负责对字符串进行操作处理。就像你们软件公司，开发部和产品部是一对搭档，程序员们从产品部获取客户需求信息，然后根据这些信息开发产品。"

11.1　C#字符串

字符串是一种基本的数据类型，在现实项目中的使用比较广泛。在本节的内容中，将对 C#字符串的知识进行简要介绍。

11.1.1 C#类型

C#支持两种字符串格式，分别是规则字符串和逐字字符串。

1．规则字符串

规则字符串是由包含在双引号中的零个或多个字符组成的，并且组成的字符可以是简单转义字符、十六进制转义字符和 Unicode 转义序列。

2．逐字字符串

逐字字符串是由字符“@”和其后面的双引号内的零个或多个字符组成的，例如@“Thanks”。在逐字字符串中，除引号外的分隔符之间的字符被全部解释。在逐字字符串中不能处理简单转义字符、十六进制转义字符和 Unicode 转义序列。

例如，下面的代码定义了一些字符串：

```
string a=”hello,MM”
string b=@”hello,MM”
string c=”hello \t MM”
string d=”hello \t MM”
string e=@”hello \t MM”
```

其中，字符“\t”是空格转义字符。

11.1.2 string 和 String

在 C#中，可以使用关键字 string 来声明一个字符串，C#中的 string 和 System.String 是完全等价的。字符串是 Unicode 字符的连续集合，可以表示文本。System.String 是 System.Char 对象的连续集合，String 的值构成该连续集合的内容。

一旦产生 String 值后就不能修改了，所以修改 String 的方法实际上是返回一个包含修改后内容的新 String。在 System.String 类中，定义了多个方法和属性来实现具体的功能，具体如下。

（1）字段 Empty

字段 Empty 表示为空字符串，是一个只读字段。

（2）String 属性

String 有 Chars 和 Lengh 属性，具体说明如下：

1）属性 Chars，用于获取实例中位于指定字符位置的字符。

2）属性 Lengh，用于获取此实例中的字符数。

（3）String 的常用方法

方法是 String 的核心，String 可以通过其本身的方法函数，实现功能强大的字符串处理。String 中常用的方法如表 11-1 所示。

表 11-1 String 的常用方法列表

方　法	描　述
Clone()	返回对此 String 实例的引用
Compare()	比较这两个 String 对象
CompareOrdial()	通过计算每个字符串中对应 Char 对象的数值来比较 String

（续）

方　　法	描　　述
CompareTo()	将此实例和指定对象或 String 进行比较
Concat()	连接 String 的一个或多个实例
Contains()	返回指定字符在某 String 对象存在的值
Copy()	复制一个指定的 String
CopyTo()	将指定字符复制到 Unicode 字符数组中的指定位置
EndsWith()	验证 String 实例结尾是否和指定字符相同
Equals()	确定两 String 是否相同
Format()	将进行格式化处理
GetEumerator()	检索可循环访问此字符串中的每个字符对象
GetTypeCode()	返回 String 的 TypeCode
IndexOF()	检索出指定字符在指定 String 对象内的第一次位置
IndexOFAny()	检索出指定字符在 Unicode 数组中任意字符对象内的第一次位置
Insert()	在指定索引位置插入一个指定 String 实例
Intern()	检索程序对 String 的引用
IsInterned()	检索程序对指定 String 的引用
IsNormallized()	设置指定字符串是否符合特定 Unicode 范式
IsNullOrEmpty()	设置某字符串是空引用还是 Empty 字符串
Join()	在指定 String 数组元素之间串联指定分隔符 String
LastIndexOf()	检索出指定字符在指定 String 对象内的最后一次位置
LastIndexOfAny()	检索在 Unicode 数组中任意字符对象内的最后一次位置
Normalize()	返回一个二进制表示的、形式符合特定 Unicode 范式的新字符串
op_Equality()	确定两指定 String 对象是否值相同
op_Inequality()	确定两指定 String 对象是否值不相同
PadLeft()	右对齐指定字符串
PadRight()	左对齐指定字符串
Remove()	删除某字符串的指定字符
Replace()	用指定字符替换指定字符串的指定字符
Split()	设置分隔某字符串的分隔符号
StarsWith()	验证 String 实例开头是否与指定字符串相同
Substring()	从此实例检索子字符串
ToCharArray()	将此实例字符复制到 Unicode 字符数组
ToLower()	返回此 String 转换为小写字符形式
ToLowerInvariant()	返回此 String 转换为小写字符形式副本，返回时使用固定区域的大、小写规则
ToUpper()	返回此 String 转换为大写字符形式
ToUpperInvariant()	返回此 String 转换为大写字符形式副本，返回时使用固定区域的大、小写规则
Trim()	从指定实例的开始和结尾处删除和指定字符
TrimEnd()	从指定实例的结尾处删除和指定字符
TrimStart()	从指定实例的开始处删除和指定字符

在接下来的内容中，将对 String 的上述主要方法进行简要说明。

1．连接字符串和提取字符串

通过连接字符串可以实现指定字符串的连接，通过提取字符串可以从字符串中将指定字符串提取出来。看下面的一段代码：

```
namespace lian1{
    class Program {
        static void Main(string[] args)
        {
            string aa = "abcdef";
            string bb = "ghijk";
            aa += bb;
            System.Console.WriteLine(aa);
            aa = aa.Substring(2, 5);
            System.Console.WriteLine(aa);
            Console.ReadKey();
        }
    }
}
```

在上述代码中，分别定义了 string 类型字符串 aa 和 bb，并设置 aa 的值是 aa 和 bb 初始值的联合。然后通过提取字符串方法 Substring()，实现了从 aa 指定位置的提取处理。程序执行后的具体结果如图 11-1 所示。

```
abcdefghijk
cdefg
```

图 11-1　实例执行结果

也可以使用 Concat()和 Join()方法来连接上述的字符串，具体代码如下：

```
namespace lian1{
    class Program{
        static void Main(string[] args) {
            string aa = "abcdef";
            string bb = "ghijkl";
            Console.WriteLine(String.Concat(aa,' ',bb));
            Console.ReadKey();
        }
    }
}
```

在上述代码中，通过 Concat()方法实现了字符串的连接。程序执行后的具体结果如图 11-2 所示。

```
abcdef ghijkl
```

图 11-2　实例执行结果

2．分隔字符串

分隔字符串方法 Split()的功能是，分隔指定数组内的字符串数据。看下面的一段代码：

```
namespace fenge{
    class Program {
```

```
            static void Main(string[] args) {
                string aa = "我 是 的 的";
                char[] separator = { ' ' };
                string[] bb;
                bb = aa.Split(separator);
                foreach (string word in bb)
                {
                    Console.WriteLine("{0}\n", word);
                }
                Console.ReadKey();
            }
        }
    }
```

在上述代码中，定义了字符串 aa 和数组 bb，然后通过字符串方法 Split()将数组内的数据进行了分隔处理。程序执行后的具体结果如图 11-3 所示。

```
file:///E:/机械工业/c#/daima/11/fenge/bin/Debug/fenge.EXE
我

是

的

的
```

图 11-3　实例执行结果

3．字符串大小写处理

使用字符串的字符大、小写处理方法，可以将指定字符串的字符转换成大写或小写。看下面的代码：

```
    namespace daxiao
    {
        class Program
        {
            static void Main(string[] args)
            {
                string aa = "My name AAAAA";
                System.Console.WriteLine(aa.ToUpper());
                System.Console.WriteLine(aa.ToLower());
                Console.ReadKey();
            }
        }
    }
```

在上述代码中定义了字符串 aa，然后分别通过方法 ToUpper()将数组内的字符进行了大写转换，通过方法 ToLower()将数组内的字符进行了小写转换。程序执行后的具体结果如图 11-4 所示。

```
file:///E:/机械工业/c#/daima/11/daxiao/bin/Debug/daxiao.EXE
MY NAME AAAAA
my name aaaaa
```

图 11-4　实例执行结果

4. 字符串比较

使用字符串的比较运算符“==”和“!=”，可以比较指定的字符串。例如，下面的代码实现了对指定字符串的比较：

```
namespace bijiao{
    class Program
    {
        static void Main(string[] args)
        {
            string aa = "我是AA";
            string bb = "我 是 A A";
            string cc = "我是AA";
            char[] separator = { ' ' };
            if (aa==bb)
            {
                System.Console.WriteLine("两者相同");
            }
            if (aa != bb)
            {
                System.Console.WriteLine("两者不相同");
            }
            if (aa == cc)
            {
                System.Console.WriteLine("两者相同");
            }
            Console.ReadKey();
        }
    }
}
```

在上述代码中，分别定义了字符串 aa、bb 和 cc，其中 aa 和 cc 相同。然后通过比较运算符“= =”和“! =”对上述字符串进行了比较处理。程序执行后的具体结果如图 11-5 所示。

file:///E:/机械工业/c#/da
两者不相同
两者相同

图 11-5　实例执行结果

另外，字符串对象内的 CompareTo()方法能够比较指定的字符串。看下面的一段代码：

```
namespace bi{
    class Program
    {
        static void Main(string[] args)
        {
            string aa = "我是AA";
            string bb = "我 是 A A";
            char[] separator = { ' ' };
            if (aa.CompareTo(bb) > 0)
            {
```

```
                        System.Console.WriteLine("aa 大于 bb");
                    }
                    else
                    {
                        System.Console.WriteLine("aa 小于等于 bb");
                    }
                    Console.ReadKey();
                }
            }
        }
```

在上述代码中，分别定义了字符串 aa 和 bb，并且 aa 和 bb 的不相同。然后通过运算符比较方法 CompareTo()对上述字符串进行了比较处理。程序执行后的具体结果如图 11-6 所示。

图 11-6　实例执行结果

5．字符串搜索

通过字符串处理方法可以从指定字符串内检索出指定字符。如果没有检索到指定的字符，则返回-1；反之则返回目标字符的位置。看下面的一段代码：

```
namespace jiansuo{
    class Program
    {
        static void Main(string[] args)
        {
            string aa = "我是 AA";
            System.Console.WriteLine(aa.IndexOf("是"));
            System.Console.WriteLine(aa.IndexOf("AA"));
            Console.ReadKey();
        }
    }
}
```

在上述代码中，设置了字符串 aa 的值是“我是 AA”，然后通过字符检索方法 IndexOf()检索 aa 内的字符“是”和“AA”。程序执行后将显示检索到的上述两字符的位置，具体结果如图 11-7 所示。

图 11-7　实例执行结果

6．字符串复制

通过字符串处理方法 Copy()和 CopyTo()，能够将某字符串复制到另一个指定字符串或 Char 类型的数组。看下面的一段代码：

```
namespace fuzhi{
    class Program
    {
        static void Main(string[] args)
        {
            string aa = "abc";
            string bb = "ede";
```

```
                Console.WriteLine("aa='{0}'",aa);
                Console.WriteLine("bb='{0}'",bb);
                Console.WriteLine("\n将aa复制到bb后\n");
                bb = String.Copy(aa);
                Console.WriteLine("aa='{0}'", aa);
                Console.WriteLine("bb='{0}'", bb);
                Console.ReadKey();
            }
        }
    }
```

在上述代码中分别设置了字符串 aa 和 bb，并各自赋予了不同的值，然后通过 WriteLine 输出了上述变量的值；通过字符复制方法 Copy()将 aa 的值复制到了 bb 中，然后通过 WriteLine 输出了上述变量的值。程序执行后将显示复制后的字符串值，具体结果如图 11-8 所示。

```
aa='abc'
bb='ede'

将aa复制到bb后

aa='abc'
bb='abc'
```

图 11-8　实例执行结果

7．字符串修改

通过 C#的字符串修改方法，可以为某字符串的值进行插入、删除、移位等处理，并且可以对字符串的某特定部分进行修改处理，处理后会返回一个新的字符串。

实例 51：使用字符串修改方法

下面通过一个简单实例来说明修改字符串的具体流程。本实例保存在“光盘：\daima\11”文件夹内，项目名为 modify。实例的功能是根据输入的内容，输出显示修改后的文本。实例文件的主要代码如下：

```
namespace modify {
    class Program
    {
        static void Main(string[] args)
        {
        string animal1 = "我";
        string animal2 = "你";
        string strTarget = "我是你的什么";
        Console.WriteLine("原字符串:{0}", strTarget);
        Console.Write("请输入一个（或一组）修饰{0}的形容词: ==> ", animal1);
        string adj1 = Console.ReadLine();
        Console.Write("请输入一个（或一组）修饰{0}的形容词: ==> ", animal2);
        string adj2 = Console.ReadLine();
        adj1 = adj1.Trim() + " ";
        adj2 = adj2.Trim() + " ";
        strTarget = strTarget.Insert(strTarget.IndexOf(animal1), adj1);
        strTarget = strTarget.Insert(strTarget.IndexOf(animal2), adj2);
        Console.WriteLine("结果字符串:{0}", strTarget);
        strTarget = strTarget.PadLeft(strTarget.Length + 5, '+');
        strTarget = strTarget.PadRight(strTarget.Length + 5, '-');
        Console.WriteLine("扩充后字符串:{0}", strTarget);
```

```
            Console.ReadKey();
            }
        }
    }
```

上述实例代码的设计流程如下。

1）分别定义字符串 animal1、animal2 和 strTarget。

2）通过 WriteLine 输出 strTarget 的值。

3）通过 ReadLine()获取第一次输入的值，并将值赋给 adj1。

4）通过 ReadLine()获取第二次输入的值，并将值赋给 adj2。

5）通过 Trim()方法将 adj1 和 adj2 的值进行移位处理。

6）通过 Insert()方法将 adj1 和 adj2 的值进行添加字符处理。

7）通过方法 PadLeft()和 PadRigh()分别对 adj1 和 adj2 进行对齐处理。

8）输出最后的处理结果。

执行后将分别输出显示对应结果，如图 11-9 所示。

```
原字符串:我是你的什么
请输入一个（或一组）修饰我的形容词: ==> youlenai
请输入一个（或一组）修饰你的形容词: ==> hao
结果字符串:youlenai 我是hao 你的什么
扩充后字符串:+++++youlenai 我是hao 你的什么-----
```

图 11-9　实例执行结果

8．字符串格式化

使用 C#中的 Format()方法，可以格式化处理某字符串内的指定字符（例如将字符替换为数字、日期、数字和时间等格式）。在下面的代码中，使用 Format()创建了一个使用整形变量的字符串。

```
    class Program {
          static void Main(string[] args) {
             int mm = 16;
             int nn = 34;
             string zz = String.Format("现在世界杯{0}强已经出现{1}支球队，当前
    时间是{2,u}", mm, nn, DateTime.Now);
             Console.WriteLine(zz);
          }
       }
```

11.1.3　StringBuilder

在 C#中，StringBuilder 表示可变字符串的类 ，它在 System.Text 命名空间内被定义。StringBuilder 通过本身定义的方法和属性来实现具体的功能，在下面的内容中将详细介绍。

StringBuilder 的常用属性如表 11-2 所示。

表 11-2 StringBuilder 的常用属性列表

属　　性	描　　述
Capacity	设置可包含当前实例所分配的内存中的最大字符数
Chars	获取实例中指定位置的字符
Length	获取当前 StringBuilder 对象的长度
MaxCapacity	获取此实例的最大容量

StringBuilder 的常用方法如表 11-3 所示。

表 11-3 StringBuilder 的常用方法列表

属　　性	描　　述
Append()	在实例结尾追加指定对象字符串的表示形式
AppendFormat()	向实例中追加多个规范、格式化的字符串
AppendLine()	将默认行终止符追加到此实例结尾
CopyTo()	将指定实例字符复制到目标 Char 数组的某字段
EnsureCapacity()	保证此 StringBuilder 的容量至少是指定值
Equals()	判断两个 StringBuilder 对象是否相等
Insert()	将指定对象字符串插入到指定对象内
Remove()	在某对象内删除指定字符串
Replace()	将某对象内的指定字符替换
ToString()	将 StringBuilder 值转换为 String

实例 52：使用 StringBuilder 处理字符串

下面通过一个简单实例来说明使用 StringBuilder 的具体流程。本实例保存在“光盘：\daima\11”文件夹内，项目名为 UesStringBuilder。实例的功能是根据定义的字符串进行各种操作处理。实例文件的主要代码如下：

```
namespace UesStringBuilder{
    class Program {
        static void Main(string[] args)
        {
            // 创建一个能够容纳 50 个字符的 StringBuilder 对象
            // 并将它初始化为"ABC".
            StringBuilder mm = new StringBuilder("ABCD", 50);
            // 将三个字符 (D, E 和 F) 添加到 StringBuilder 对象末尾
            mm.Append(new char[] { 'D', 'E', 'F' });
            // 将一个格式字符串添加到 StringBuilder 对象末尾
            mm.AppendFormat("GHI{0}{1}", 'j', 'K');
            // 显示 StringBuilder 对象中字符数和整个字符串
            Console.WriteLine("{0} 个字符: {1}\n", mm.Length, mm.ToString());
            // 将一个字符串插入到 StringBuilder 对象的开头
            mm.Insert(0, "Myname: ");
            // 用大写 J 替换字符串中的小写 j
```

```
                mm.Replace('j', 'J');
                // 显示 StringBuilder 对象中字符数和整个字符串
                Console.WriteLine("{0} 个字符: {1}\n", mm.Length, mm.ToString());
                // 删除最开始的两个字符
                mm.Remove(0, 2);
                // 显示 StringBuilder 对象中字符数和整个字符串
                Console.WriteLine("{0} 个字符: {1}\n", mm.Length, mm.ToString());
                Console.ReadKey();
            }
        }
    }
```

上述实例代码的设计流程如下。

1）定义 StringBuilder 字符串 mm。

2）将字符串内的字符“D”、“E”和“F”进行追加处理。

3）通过 AppendFormat()方法追加多个格式字符。

4）通过 Insert()方法进行添加处理。

5）通过 Replace()方法对字符“j”进行替换处理。

6）通过 Insert()方法将 adj1 和 adj1 的值进行添加字符处理。

7）通过 Remove()方法删除字符串内的指定字符。

8）输出最后的处理结果。

执行后将分别输出显示对应结果，如图 11-10 所示。

```
file:///E:/机械工业/c#/daima/11/UesStringBuilder
12 个字符: ABCDDEFGHIjK

20 个字符: Myname: ABCDDEFGHIJK

18 个字符: name: ABCDDEFGHIJK
```

图 11-10　实例执行结果

11.2　C#正则表达式

使用正则表达式，可以灵活迅速地实现功能强大的文本处理。正则表达式可以快速分析指定文本，并找到特定的字符模式，从而实现提取、编辑、替换和删除指定字符串的功能。在现实应用中，正则表达式被广泛应用在 HTML 处理、日志文件等应用程序中。

在.NET 框架中的 System.Text.RegularExpressions 命名空间中，定义了专用的正则表达式类库。正则表达式类库可以和面向公共语言运行时的任何语言或工具一起使用。在.NET 框架中，正则表达式类库支持如下三种基本应用。

1）分隔符分隔处理。

2）字符检索处理。

3）字符替换处理。

11.2.1 语言

正则表达式的语言是正则表达式的核心，它可以将文档内要搜索的文本、转义序列和特定字符组合在一起，这样就能实现特定需求的字符检索。

正则表达式语言由原义文本字符和元字符这两种字符类型组成。其中，原义文本字符是某字符的原始实际字符；元字符是一些处理字符，能够实现特定功能的处理。正则表达式元字符由五种类型构成，接下来将一一讲解这五种类型。

1．转义符类型

转义符类型可以将文本内的下一个字符转换为特殊字符。C#正则表达式内只有一个转义字符，即“\”。例如，“n”匹配字符“\n”，而“n\”和换行符相匹配。

2．定位点类型

定位点类型可以对文本内的指定字符进行定位处理。在 C#正则表达式内有多个定位点类型，具体说明如表 11−4 所示。

表 11−4　定位点类型列表

字　　符	描　　述
/\	用于匹配输入字符串的开始位置
$	用于匹配输入字符串的结尾位置
\A	用于匹配输入字符串的开头的位置
\b	用于匹配字符串内的空白字符
\B	用于匹配字符串内的非空白字符
\G	设置匹配必需出现在上一个匹配结束的地方
\z	用于匹配输入字符串的结尾位置
\Z	用于匹配输入字符串的结尾和换行符之前位置

3．限定符类型

限定符类型的功能是，对文本内的指定字符进行限定处理。在 C#正则表达式内有多个限定符类型，具体说明如表 11−5 所示。

表 11−5　限定符类型列表

字　　符	描　　述
*	用于指定 0 个或多个匹配
+	用于指定 1 个或多个匹配
?	用于指定 0 个或 1 个匹配
{n}	用于指定正好 n 个匹配
{n,}	用于指定至少 n 个匹配
{n,m}	用于指定大于 n 但小于 m 个匹配
*?	设置尽可能少的使用重复的第一个匹配
+?	设置尽可能少的使用重复但至少使用 1 次
??	设置使用 0 次或 1 次重复
{n}?	和{n}(lazy{n})等效
{n,}?	设置尽可能少的使用重复但至少使用 n 次
{n,m}?	设置尽可能少的使用重复，但要使用 n～m 次

4．匹配模式类型

匹配模式类型的功能是，对文本内的指定字符进行各种模式的匹配处理。在 C#正则表达式内有多个匹配模式类型，具体说明如表 11-6 所示。

表 11-6　匹配模式类型列表

字　　符	描　　述
?	当此字符紧跟在限定符之后时，匹配模式检索到尽可能短的字符串
.	用于匹配除“\n”之外的任何单个字符
(patterm)	用于匹配 patterm 并捕获该匹配的子表达式
(?:patterm)	用于匹配 patterm 但是不捕获该匹配的子表达式
(?=patterm)	执行正向预测搜索表达式，此表达式的匹配位置是 patterm 字符串的起始点的搜索字符串
(?!patterm)	执行反向预测搜索表达式，此表达式的匹配位置不是 patterm 字符串的起始点的搜索字符串

5．字符匹配类型

字符匹配类型的功能是，对文本内的指定字符设置匹配的字符。在 C#正则表达式内有多个字符匹配类型，具体说明如表 11-7 所示。

表 11-7　字符匹配类型列表

字　　符	描　　述
x|y	用于匹配 x 或 y
[xyz]	用于匹配包含的字符
[∧xyz]	用于匹配未包含的字符
[a-z]	用于匹配指定范围内的字符
[a-z]	用于匹配指定范围内没有的字符
\cx	用于匹配 x 指示的字符
\d	数字字符匹配
\D	非数字字符匹配
\f	换页符匹配
\n	换行符匹配
\r	回车符匹配
\s	空白符匹配
\S	非空白符匹配
\t	制表符匹配
\v	垂直制表符匹配
\w	任何字类字符匹配
\W	任何非字类字符匹配
\xn	匹配其中的十六进制转义码 n
\num	匹配正整数 num
\n	标识一个八进制转义码或反向引用
\nm	标识一个八进制转义码或反向引用
\nm1	当 n 是八进制数 0～3，m 和 1 是八进制数 0-7 时，匹配八进制转义码 1
\un	匹配四位十六进制数表示的 Unicode 字符 n

正则表达式的模式设置在文本搜索时要匹配的一个或多个字符串，例如，下面的代码就是用于验证两位数字、一个连接字符和五位数字组成的 ID 号的：

```
\d{2}-\d{5}
```

11.2.2 正则表达式类

在 System.Text.RegularExpressions 命名空间内，定义了七个常用的正则表达式类。分别是 Regex、Match、MatchCollection、GroupCollection、CaptureCollection、Group 和 Capture。上述各个类对象之间的关系如图 11-11 所示。

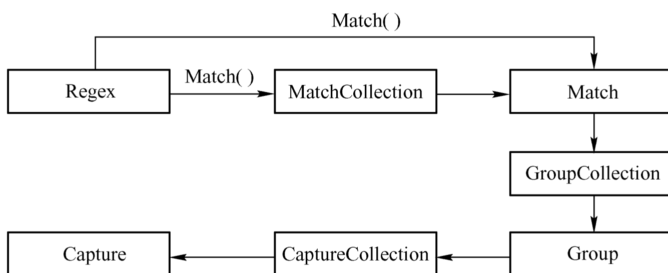


图 11-11 正则表达式类对象的关系

在下面的内容中，将分别简要介绍上述正则表达式类的具体应用方法。

1．Regex

Regex 表示只读的正则表达式，这样的正则表达式是不可变的。在 Regex 内使用了各种静态方法，允许不显式的创建其他类的新实例，以使用其他正则表达式类。

Regex 类中有如下四种方式的构造函数。

- Regex()：用于初始化 Regex 类的新实例。
- Regex(String)：用于为指定的正则表达式初始化并编译 Regex 类的一个新实例。
- Regex(SerializationInfo,StreamingContext)：用序列化数据初始化 Regex 类的新实例。
- Regex(String,RegexOptions)：用修改模式选项为指定的正则表达式初始化并编译 Regex 类的一个新实例。

在 Regex(String,RegexOptions)方式中，可以在构造函数内传入一个 RegexOptions 枚举值，用于设置正则表达式的选项。RegexOptions 定义值的具体说明如下。

- Compiled：设置将正则表达式编译为程序集。
- CultureInvariant：设置忽略语言中的区域性差异。
- ECMAScript：设置表达式启用符合 ECMAScript 的行为。
- ExplicitCapture：设置有效捕获仅形式为(?<name>…)的显式命名或编号的组。
- IgnoreCase：设置不区分大、小写的匹配。
- IgnorePatternWhitespce：消除模式中的非转义空白，并启用使用#标记的注释。
- Multiline：多行模式，用于更改∧和$的含义。
- None：指定不设置选项。
- RightToLeft：设置搜索从右向左进行。

- SingleLine：设置为单行模式。

2．Match

Match 类表示正则表达式匹配操作的结果。使用 Regex 类的 Match()方法，可以返回 Match 类型对象，以便找到输入字符串中的第一个匹配项。例如下面的代码：

```
Regex mm=new Regex(“abc”);
Match nn=mm.Match(“345abc123”);
 if (nn.Success)
      Console.WriteLine("匹配的位置是：", nn.Index);
```

在上述代码中，使用了 Match 类的 Match. Success 属性来验证是否找到匹配。

3．MatchCollection

MatchCollection 类用于表示成功的、非重叠的匹配序列。此类是只读的，没有公共构造函数。MatchCollection 实例由 Regex.Matches()方法返回。看下面一段代码：

```
namespace Matchchuli{
    class Program
    {
        static void Main(string[] args)
        {
            Regex mm = new Regex("abc");
            MatchCollection nn = mm.Matches("321abc456cdef33abc333abcdef");
            String[] shuzu = new String[20];
            int[] zu = new int[20];
            for (int i = 0; i < nn.Count; i++)
            {
                shuzu[i] = nn[i].Value;
                zu[i] = nn[i].Index;
                Console.WriteLine("shuzu[{0}]=\"{1}\",zu[{2}]={3}",i,shuzu
[i],i,zu[i]);
            }
            Console.ReadKey();
        }
    }
}
```

在上述代码中，通过使用 Regex 类的 Matches()方法在指定字符串中找到指定字符的匹配位置。具体实现流程如下。

1）定义 Regex 类对象 mm，用于设置检索字符是“abc”。

2）设置检索对象 nn 的值是“321abc456cdef33abc333abcdef”。

3）使用 Matches()方法开始检索。

4）将检索结果保存在数组 shuzu 和 zu 内。

5）通过 WriteLine()方法逐一输出检索结果。

执行后将输出字符串“321abc456cdef33abc333abcdef”内的字符“abc”的位置，如图 11-12 所示。

```
shuzu[0]="abc",zu[0]=3
shuzu[1]="abc",zu[1]=15
shuzu[2]="abc",zu[2]=21
```

图 11-12　实例执行结果

4．GroupCollection

GroupCollection 类能够在捕获某组集合的同时，返回单个匹配中捕获的组的集合。此集合是只读的，没有公共构造函数。GroupCollection 类的实例在 Match.Groups()方法中返回。

例如，通过下面的代码查找并输出由正则表达式捕获的数组数目：

```
namespace Groupchuli{
    class Program
    {
        static void Main(string[] args)
        {
            Regex mm = new Regex("(a(b))c");
            Match nn = mm.Match("abdabcdcbabccscab");
            Console.WriteLine("找到组的数量是：{0}", + nn.Groups.Count);
            Console.ReadKey();
        }
    }
}
```

在上述代码中，定义了 Regex 类实例对象 mm，设置了“abc”、“ab”和“b”的组。然后使用 Match 属性，返回了 nn 内检索到上述组的数量。具体执行结果如图 11-13 所示。

```
找到组的数量是：3
```

图 11-13　实例执行结果

5．CaptureCollection

CaptureCollection 类可以设置捕获的子字符串序列，并返回由单个捕获组执行的集合。一个捕获组可以在单个匹配中捕获多个字符串。其中 Math 和 group 共同提供 Captures 属性，以便于对捕获的子字符串集合的访问。看下面的一段代码：

```
namespace Capturechuli{
    class Program
    {
        static void Main(string[] args)
        {
            String ss = "ABCDABCDabcdABcd";
            Console.WriteLine("字符串：{0}", ss);
            Regex rr = new Regex("(AB)+");
            Match mm = rr.Match(ss);
            GroupCollection gg = mm.Groups;
            Console.WriteLine("捕获到的数组：{0}", gg.Count);
            int counter;
            CaptureCollection cc;
            for (int i = 0; i < gg.Count; i++)
```

```
                {
                    cc= gg[i].Captures;
                    counter = cc.Count;
                    Console.WriteLine("捕获到的数目：{0}", counter);
                }
                Console.ReadKey();
            }
        }
    }
```

在上述代码中，使用了正则表达式“(AB)+”来检索字符串“ABCDABCDabcdABcd”中的一个或多个匹配，然后使用了 Captures 属性返回了多组捕获的子字符串。具体执行结果如图 11-14 所示。

```
字符串：ABCDABCDabcdABcd
捕获到的数组：2
捕获到的数目：1
捕获到的数目：1
```

图 11-14　实例执行结果

6．Group

Group 类用于表示来自单个捕获组的结果。因为 Group 可以在单个匹配中捕获 0 个、1 个或多个字符串，所以里面也包含 Captures 对象的集合。并且因为 Group 继承于 Capture，所以可以直接访问最后捕获的子字符串。

Group 实例有如下两种返回情况。

1）Match.Groups(groupnum)属性返回。

2）Match.Groups(“groupname”)属性返回，此时使用“(?<groupname >)”分隔符。

看下面的一段代码：

```
    namespace Groupchuli1{
        class Program
        {
            static void Main(string[] args)
            {
                Regex rr = new Regex("(a(b))c");
                Match mm = rr.Match("abdabcabcabd");
                int[] aa = new int[20];
                String[] ss = new String[20];
                for (int i = 0; mm.Groups[i].Value!= ""; i++)
                {
                    ss[i] = mm.Groups[i].Value;
                    aa[i] = mm.Groups[i].Index;
                    Console.WriteLine("捕获到的结果：ss[{0}]=\"{1}\",aa[{2}]
    ={3}\n", i ,ss[i],i,aa[i]);
                }
                Console.ReadKey();
            }
        }
    }
```

在上述代码中，分别定义了检索对象 rr 的值是检索字符串“abc”、“ab”和“b”的组；

然后通过嵌套分组构造将子字符串捕获到组中。具体执行结果如图 11-15 所示。

```
捕获到的结果：ss[0]="abc",aa[0]=3
捕获到的结果：ss[1]="ab",aa[1]=3
捕获到的结果：ss[2]="b",aa[2]=4
```

图 11-15　实例执行结果

7．Capture

Capture 类包含来自单个子表达式的捕获结果。例如在下面代码中，从 Group 中的每个成员中提取 Capture 集合，并且将变量 pp 和 ll 分配给找到的每一个字符串的初始字符串中的位置。

```
static void Main(string[] args){
          Regex rr = new Regex("(ab)*");
          Match mm = rr.Match("bcabcabc");
          CaptureCollection cc;
          int pp, ll;
          for (int i = 0; mm.Groups[i].Value != ""; i++)
          {
             cc = mm.Groups[i].Captures;
             for (int j = 0; i < cc.Count; j++)
             {
                pp = cc[j].Index;
                ll = cc[j].Length;
             }
          }
          Console.ReadKey();
       }
```

11.3　正则表达式应用

在.NET 框架中的正则表达式类库支持如下三种基本应用。

- ❑ 分隔符分隔处理。
- ❑ 字符检索处理。
- ❑ 字符替换处理。

接下来将一一讲解上述三种基本应用的知识。

11.3.1　分隔符分隔

通过分隔符能够将制定的字符串实现拆分处理。通过 Regex 类内的 Split()方法，能够实现字符串拆分功能。Regex 类提供了五种不同形式的 Split()方法，其中有两种是静态方法。静态 Split()方法能够独立、单独的使用正则表达式，不用显式的创建 Regex 对象。看下面的

一段代码：

```
namespace chaifen1{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("输入一段文字（按 Enter 结束）：");
            string s = Console.ReadLine();
            string[] ss = Regex.Split(s, @"\.|,|;|:|。|，|；|：");
            Console.WriteLine("\n==>将你输入的句子按照上述分隔符进行分割处理,处
理后的结果是：");
            foreach (string str in ss)
                Console.WriteLine(str);
            Console.ReadKey();
        }
    }
}
```

在上述代码中，根据 ss 内的分隔符对输入控制台的字符串进行分隔处理。例如，输入“aaaaaaaaaa,bbbbb.ccccc”后的处理结果如图 11-16 所示。

```
输入一段文字（按Enter结束）：aaaaaaaaaa,bbbbb.ccccc

==>将你输入的句子按照上述分隔符进行分割处理,处理后的结果是：
aaaaaaaaaa
bbbbb
ccccc
```

图 11-16　实例执行结果

11.3.2　字符检索处理

正则表达式能够在指定字符串内检索匹配特定模式的子字符串。Regex 类使用如下三个方法来实现字符串的检索处理。

1）Matches，在 Matches 对象中包含一个 Groups 属性，其中包含了匹配得到的子字符串。

2）IsMatch，标识正则表达式在输入字符串中是否找到匹配项。例如，下面的代码用于检查字符串中是否包含标记“<a>”：

```
Regex rr=new Regex(“<a[ >]*>”,RegexOptions.IgnoreCase);
if (regex.IsMatch(s))
  Console.WriteLine("字符串{}包含标记<a>");
Else
  Console.WriteLine("字符串{}不包含标记<a>");
```

另外，IsMatch 还可以验证用户的输入。

3）Match，在指定的输入字符串中搜索 Reges 构造函数中指定的正则表达式匹配项，并

返回一个 Match 类对象。例如在下面的代码中，显示了 HTML 片段中所有带引号的 URL：

```
void chuli(string s)
{
    Regex rr =new Regex("href\\s*=\\s*\"[ \"]*\"",RegexOptions.IgnoreCase);
    Match mm=rr.Match(s);
    while (mm.Success)
    {
    Console.WriteLine(mm.Value);
        mm=mm.NextMatch();
    }
}
```

11.3.3 字符替换处理

使用 Regex 类中的 Replace()方法，可以替换指定的字符串。例如在下面代码中，将“我很高兴”替换为“我很好”：

```
Regex rr =new Regex("我很高兴”);
string soutput = regex. Replace(sinput,”我很好”);
```

和 Split()方法类似，Replace()也有静态方法的形式。静态 Replace()方法和使用指定的正则表达式模式构造 Regex 对象等效，并能调用 Replace 实例方法。使用静态 Replace()方法能够便于独立、单独的使用正则表达式，而不必显式的创建 Regex 对象。

11.4 C# 4.0 新特性——命名可选参数与 COM 互操作

以前，C# 2.0 和 Java 一样，是一门的纯粹的面向对象的语言，都使用重载而不是可选参数。但是实际上使用的其他外部程序，COM 组件却经常不要求指定所有参数。这会导致一个 C#程序员不得不用 Type.Missing 来填满整个参数列表。幸好从 C#4.0 开始支持命名参数/可选参数了。程序员可以在方法调用的时候通过命名参数指定可选参数。而这一切都是为了让.NET 4.0 的动态语言运行库(DLR)在动态绑定的时候具有更好的兼容性。

11.4.1 命名与可选参数

这是 C# 4.0 的一个新特征，但对 C++、Visual Basic 和 Python 等程序员来说，这只是很普通的一个特征。C# 4.0 中的可选参数和其他语言中的用法大致相同，这里不需要额外的关键词修饰，例如下面的声明是合法的：

```
static void Foo(int a, String s = "i'm a string", dynamic b =null,
MyClass c = null)
```

在 C#4.0 中使用可选参数时，必须遵循以下几条原则。

1）可选参数必须有编译时常量作为其默认值。如果是除 String 之外的引用类型(包括特殊的 dynamic 类型)，默认值只能是 null。下面的声明是不能通过编译的：

```
static void Foo(int a, String s = "i'm a string", dynamic b = 2,
MyClass c = new MyClass())
```

2）可选参数必须从右往左出现在参数列表中(必须后出现)，可选参数右边的参数(如果有的话)必须是可选参数。下面的声明是不能通过编译的：

```
static void Foo(String s = "i'm a string", int a, dynamic b = null,
MyClass c = null)
```

3）可选参数的赋值必须通过命名参数的方式指定，即必须使用可选参数的参数名称对其进行赋值。而非可选参数则不一定要用命名参数。如下的调用都是合法的：

```
Can calls
Can(2);
Can(a: 2);
Can(2, s: "hello", b: 3.14);
Can(2, s: "hello", b: 3.14, c:new MyClass());
Can(2, c: new MyClass());
```

一旦调用方法时使用的是命名参数，则命名参数的位置可以是任意顺序，如下的调用是合法的：

```
Can(b:3.14, c:new MyClass(), a:2);
```

唯一的影响就是上述调用中会对 b 先求值，然后再是 c 和 a。函数总是按照参数出现的顺序进行求值操作的。另外，可选参数不仅适用于普通的方法，还适用于构造器、索引器中，本质上它们没有什么不同。

11.4.2　可选参数与重载决策

虽然命名参数和可选参数使 CLR 在方法重载决策(overload resolution)方面变得更加复杂，不过作为开发人员，只需要搞清重载决策的下面几个特点就可以了。

1）在带可选参数的方法签名中，重载决策不会认可被可选参数代替的重载版本，比如下面两个声明：

```
public static void Foo(int a, String s = "i'm a string", dynamic b =
null, MyClass c = null);
public static void Foo(int a, String s = "i'm a string");
```

如果按照以下方式调用，编译器会提示它已经被上面两个方法 confused 了：

```
Foo(2);
Foo(a: 2);
```

2）在调用方式同样合法的情况下，重载决策会优先选择不带可选参数的重载版本。比如下面两个方法：

```
public static void Foo(int a, String s = "i'm a string", dynamic b =
```

```
null, MyClass c = null);
public static void Foo(int a);
```

如果使用以下方式调用，被调用的会是 void Foo(int a);这个版本：

```
Foo(2);
Foo(a: 2);
```

3）在调用方式同样合法的情况下，重载决策会优先选择类型最为匹配、最易转化的重载，例如下面两个方法：

```
public static void Foo(byte a, String s = "i'm a string", dynamic b =
null, MyClass c = null);
public static void Foo(object a);
```

Foo(2)和 Foo(a:2)都将调用前一个方法，因为 int 转到 byte 是值类型之间的转化，其代价要比从 int 转到 object 的代价低。

11.4.3 COM 互操作

动态类型绑定和命名参数，可选参数都为 CLR 的 COM 互操作提供了便利。有了动态类型，访问 COM 对象时就不必再对返回的 COM 对象进行显式的类型转换了，因为返回的是一个 dynamic 类型的对象，可以直接在它上面调用方法/属性/索引器等，例如 excelApp.Cells[row,column].Value = "some value"; 这里的 Cells[row,column]返回的是一个 dynamic 类型的对象，而之前都是返回的 object，需要显式转换才能操作它，比如：((Excel.Range)excelApp.Cells[row,column]).Value = "some value"。

有了可选参数，开发人员再也不用 Type.Missing 来填充整个参数列表了，比如 Office12 的 Excel.ApplicatioinClass 中就利用了可选参数定义了一个 intersect 方法：

```
public virtual Range Intersect(Range Arg1,
Range Arg2,
object Arg3 = Type.Missing,
object Arg4 = Type.Missing,
object Arg5 = Type.Missing, object Arg6 = Type.Missing,
object Arg7 = Type.Missing, object Arg8 = Type.Missing,
object Arg9 = Type.Missing, object Arg10 = Type.Missing,
object Arg11 = Type.Missing, object Arg12 = Type.Missing,
object Arg13 = Type.Missing, object Arg14 = Type.Missing,
object Arg15 = Type.Missing, object Arg16 = Type.Missing,
object Arg17 = Type.Missing, object Arg18 = Type.Missing,
object Arg19 = Type.Missing, object Arg20 = Type.Missing,
object Arg21 = Type.Missing, object Arg22 = Type.Missing,
object Arg23 = Type.Missing, object Arg24 = Type.Missing,
object Arg25 = Type.Missing, object Arg26 = Type.Missing,
object Arg27 = Type.Missing, object Arg28 = Type.Missing,
object Arg29 = Type.Missing,
```

```
object Arg30 = Type.Missing
);
```

而在以前的版本中，需要手动指定这 30 个参数。

在性能方面，由于 C#4.0 中编译器可以把 PIAs 按需部分编译到的程序集中，所以不必每次都完全载入这组庞大的互操作程序集，这对提高 COM 交互性能很有帮助。另外还有一个语法就是在 COM 调用的时候可以不使用 ref 传递参数，不过这里不使用 ref 并不代表不按引用传递而按值传递，实际上 C#编译器会自行帮助创建临时变量并仍然按引用把参数传递给调用者。



C#4.0 中很大一部分特征弥补了它之前的一些令开发者不满意的地方，无论是动态类型还是可选参数，新的 C#让那些和各种组件（COM、IronPython 等等）打交道的程序员获得一定程度的解脱。C#越来越变得以人为本，更确切地说，以程序员为本。

11.5　疑难问题解析

本章详细介绍了 C#中字符串和字符表达式的基本知识。本节将对本章中比较难以理解的问题进行讲解。

读者疑问：请问，使用分隔符的原则是什么？

解答：分隔符分隔的原则如下。

1）如果指定了 count 的值，则只能将字符串拆分为 count 个字符串。

2）如果 count 的值是 0，则尽可能多的进行拆分。

3）如果指定了 startat，则搜索第一个分隔符，以便从指定的位置开始执行操作。

4）如果在正则表达式使用捕获组，则生成的字符串数组中会包含捕获组。

读者疑问：择一匹配“|”的具体使用技巧是？

解答：C#正则表达式中的“|”符号似乎没有一个专门的称谓，姑且称之为“择一匹配”吧。事实上，像[a-z]也是一种择一匹配，只不过它只能匹配单个字符，而(|)则提供了更大的范围，(ab|xy)表示匹配 ab 或匹配 xy。注意“|”与“()”在此是一个整体。例如下面的代码:

```
Code
string x = "0";
string y = "0.23";
string z = "100";
string a = "100.01";
string b = "9.9";
string c = "99.9";
string d = "99.";
string e = "00.1";
Regex r = new Regex(@"^\+?((100(.0+)*)|([1-9]?[0-9])(\.\d+)*)$");
Console.WriteLine("x match count:" + r.Matches(x).Count);//1
```

```
        Console.WriteLine("y match count:" + r.Matches(y).Count);//1
        Console.WriteLine("z match count:" + r.Matches(z).Count);//1
        Console.WriteLine("a match count:" + r.Matches(a).Count);//0
        Console.WriteLine("b match count:" + r.Matches(b).Count);//1
        Console.WriteLine("c match count:" + r.Matches(c).Count);//1
        Console.WriteLine("d match count:" + r.Matches(d).Count);//0
        Console.WriteLine("e match count:" + r.Matches(e).Count);//0
```

在上述代码中，匹配了 0 到 100 的数。最外层的括号内包含两部分，即“(100(.0+)*)”、“([1-9]?[0-9])(\.\d+)*”，这两部分是“OR”的关系，即正则表达式引擎会先尝试匹配 100，如果失败，则尝试匹配后一个表达式（表示（0,100）范围中的数字）。



职场点拨——部门间的沟通

在一个公司中，总会有不同的部门，这些不同的部门能够完成不同的职责。例如程序员属于技术部或开发部，除此之外还有财务部和人力资源部等。部门与部门之间的沟通的确比较麻烦，有些事情也不受个人的掌控，虽然明知很多问题的根源，却没有力气去解决。部门之间的沟通向来是职场中的话题，在此提出如下三点经验希望对读者有用。

（1）要有一些容人之量

身边的同事，有的能力很好，也有的能力不强。如果大家自身没有广阔的心胸，那在社会的大舞台上必将撞得头破血流。要有容人之量，应该容忍这样的现状，只要尽自己最大的努力来完成自己应该做的事情即可。这样会无愧于公司，也不会耽误自己学习文化知识。在部门之间，应该主动和对方部门沟通，本着解决问题的目的来做事。

（2）各个部门的成员之间更应该互相尊重

举个例子，产品部没有了解到客户的需求，就让开发人员进行开发。到后来因为客户不满意，而重新返工修改。改来改去，体现了产品部对技术人员工作成果的不尊重。如果在出现问题的时候不承担自己应承担的责任，就更会受到研发人员的反感。站在研发人员的角度来讲，最不希望的就是干了活，受了累，却因为别人的错而重新干一遍。所以和开发人员关系最密切的产品部做沟通时，一定要使用书面材料，在上面将需求写得清清楚楚，防止出错后对方不认账。

（3）不要抱怨还是要抱怨

刚开始，大家总会有容人之量地进行着和谐的工作，当因为情绪不满而影响到工作动力和态度的时候，应该及时找到问题所在。但是在开发的时候，还是有团队成员抱怨，抱怨不公，抱怨自己的工作量大，去想怎样才能快速地完成自己的任务。此时开发人员应该将心中的种种不快和认为平时工作中不正确的地方反馈给项目经理，让他去帮助大家来解决此类问题。

第三篇　提高篇

第 12 章　委托与事件

C#中的委托是一种引用类型，它存储了对方法的引用。委托实际上是一种类型安全的、使用回调的方法函数。事件是一种重要的函数成员，类和对象使用事件来通知其他对象发生的行为或某条件已成立。事件通过使用委托后，能够封装触发时将要调用的方法。在本章的内容中，将详细介绍 C#中委托与事件的基本知识，并通过简单的实例来介绍其具体的使用流程。通过本章能学到如下知识。

- ❑ C#委托。
- ❑ 事件。
- ❑ 动态编程。
- ❑ 职场点拨——跳槽的成本。

2008 年 XX 月 XX 日，小雪

到今天为止，我已经工作一年半了，工资从当初的 3000 元涨到了 4000 元。但是总感觉工资不够花的，女友一直在催买房结婚，我感觉压力好大。不觉间慢慢产生了跳槽去寻求更好待遇的想法。



一问一答

小菜：我想找一份更好的工作，这里的待遇太低了！

Wisdom：人往高处走这无可厚非，但是你能确保一定能找到更好待遇的工作吗？

小菜：这个……

Wisdom：呵呵，你自己也不敢确定吧，建议你看下本章最后的‘跳槽的成本’，或许会给你带来一些帮助！

小菜：嗯，言归正传，本章的委托与事件有什么用？

Wisdom：你要寻找所谓另外的生存方式，当然是以赚钱为目的。本章所要讲的委托与事件也是一种新的类型，是以解决更加复杂的问题为目的。

12.1 C#委托

在 C#中使用委托后，能够处理在其他编程语言中需要使用函数指针来处理的问题。委托和函数比较类似，和函数相比主要有如下三点区别。

1）委托是匿名的。

2）委托是面向对象和类型安全的，而函数的指针是不安全的类型。

3）委托同时封装了对象实例和方法，而函数指针仅指向函数成员。

委托不会关心它所封装的方法或所属的类，它只是负责实现这些方法和委托的类型相兼容。

12.1.1 声明委托

在 C#中使用关键字 delegate 来声明委托，具体格式如下：

```
修饰符 delegate 返回类型 委托名(形参);
```

其中，委托的修饰符包括访问修饰符和 new，不能在同一个委托内多次使用同一个修饰符。修饰符 new 用于隐藏从基类继承而来的同名委托。public、protected、internal 和 private 用于控制委托类型的可访问性，但是根据具体的需要可能不允许使用某修饰符。

因为 C#委托名是一种标识符，所以需要遵循标识符的命名规则，并且最好能体现出委托的含义和用途。

形参是可选的，用来指定委托的参数。返回类型用于设置委托的返回类型。

如果一个方法和某委托相兼容，则这个方法必须具备如下两个条件。

1）两者具有相同的签名，即具有相同的参数数量，并且类型、顺序和参数修饰符也相同。

2）两者返回类型相同。

看下面的一段代码：

```
delegat int weituo(Object mm,Int i);
```

在上述代码中，声明了一个 Int 类型的委托 weituo，并且包含了 Object 类型的参数 mm，int 类型的参数 i。上述委托可以和下面代码中的方法 chuli()相兼容：

```
int chuli(Object mm,int i);
```

在 C#中，即使两个委托的名字相同，但是如果结构不同，则这两个委托也不是同一个委托。看下面的一段代码：

```
delegat int A1(double db,Iint i);
class class1,{
  public static int chuli1(double db1,int ii);
         {
         }
}
```

```
class class2,{
  delegat int A2(double db,Int iii);
public static int chuli1(double db2,Int iiii);
         {
         }
  public static int chuli2(double db3,int iiiii);
         {
         }
  public static int chuli3(int aa);
         {
         }
public static int chuli3(int bb);
         {
         }
}
```

在上述代码中，委托 A1 和 A2 都会和类 class1 的方法 chuli1()和类 class2 的方法 chuli1()相兼容，因为它们具有相同的返回类型和参数列表。但是委托 A1 和 A2 不会被认为是相同的，并且不能互换。而委托 A1 和 A2 都不会和类 class2 内的方法 chuli2()、chuli3()和 chuli4()相兼容，因为它们的返回类型和参数不同。



对于结构相同的委托在实例进行比较时，可以被认为是相等的关系。通过使用“==”和“！=”重载运算符，可以直接对委托实例进行比较。例如下面的代码:

```
A1 aa1=new A1(zz);
A2 aa2=new A2(zz);
Console.WriteLine(aa1==aa2);
```

委托的类型是从 System.Delegate 类派生来的类型，被隐含为封闭的。在 C#中，不允许从一个委托类中派生出其他的类型，也不允许从 System.Delegate 派生出非委托类的类型。

通过委托创建实例后，这个实例就封装了一个调用列表，在这个列表中包含了一个或多个方法。在此委托实例中存储了对调用列表中各个方法的引用，所以用合适的参数来调用委托的实例，和用这个参数集来顺序调用列表中的方法是等效的。看下面的一段代码：

```
public delegat int weituo(double db,Int i);
```

当编译器处理上面的委托声明时，将会在其中定义一个如下格式的类：

```
public class weituo : System.MulticastDelegat;{
    public weituo(double db1,Int ii);              //声明构造函数
    public int mm(double db1,Int ii);              //和委托的声明一致
}
```

在声明一个委托时，可以在任何位置（包括命名空间、类和编译单元等里面）。声明这个委托，

C#通过使用关键字 new 来创建一个委托实例，具体格式如下：

委托名 实例名 =new 委托名(参数);

如果委托实例的参数是一个方法，则标识的方法既可以是实例方法，也可以是静态方法，但是必须确保方法和委托相兼容。如果委托实例的参数是委托名，则设置实例创建委托副本。例如：

```
public delegat void PP(string xinxi);
public class lei{
    public void pp1(string xin);
        {
            Console.WriteLine(xin);
         }
}
public class test{
    public static void pp2(string xin);
        {
            Console.WriteLine(xin);
         }
}
public static void Main(); {
    lei ll=new lei();
    PP mm=new PP(ll.pp1);
    PP nn=new PP(pp2);
}
```

12.1.2 使用委托

当实现一个委托实例化后，可以像传递参数一样来传递这个对象。委托会把对它进行的方法调用传递给方法，调用方传递给委托的参数被传递给方法，方法的返回值由委托返回给调用方。

被委托实例所封装的方法集合被称为调用列表。当使用方法来创建委托实例时，这个委托实例会封装该方法，并将其作为调用列表内的第一个方法。

在 C#中为调用委托提供了专用的语法，具体说明如下。

- ❑ 当调用非空、调用列表仅包含一个方法的委托实例时，会调用调用列表中的这个方法，并且委托调用所使用的参数和返回值，参数和返回值分别与该方法内的相同。
- ❑ 如果在调用过程中发生异常，但是在调用方法中没有捕获到异常，则会在调用委托的方法内继续检索和该异常对应的 catch 子句。

1．创建静态方法

在具体的项目应用中，可以使用静态方法来创建委托实例。

实例 53：根据定义的实例输出对应文本

下面通过一个具体的实例来说明使用静态方法来创建委托实例的流程。本实例保存在“光盘：\daima\11”文件夹内，项目名为 useweituo。实例的功能是根据定义的实例输出对应的文本。实例文件的主要代码如下所示：

```
namespace useweituo {
    public delegate int weituo(string message);
    public class Test {
        static int chuli(string msg)
        {
            Console.WriteLine(msg);
            return msg.Length;
        }
        // 使用委托实例作为参数
        static void chuliTitle(weituo mm)
        {
            mm("==================================");
            mm("          使用委托!!!          ");
            mm("==================================");
        }
        public static void Main()
        {
            // 创建委托实例
            weituo mm = new weituo(chuli);
            string s = "直接调用委托实例";
            int n = mm(s);            // 直接调用委托实例
            chuliTitle(mm);           // 将委托实例作为参数来传递
            Console.ReadKey();
        }
    }
}
```

在上述实例代码中，定义了委托 weituo，并将委托实例 mm 当作参数进行了方法传递，然后通过调用静态方法 chuliTitle()输出了对应的文本。具体执行结果如图 12−1 所示。

```
直接调用委托实例
==================================
          使用委托！！！
==================================
```

图 12−1　实例执行结果

多学一招

委托和事件在.Net Framework 中的应用非常广泛，然而，较好地理解委托和事件对很多接触 C#时间不长的人来说并不容易。它们就像是一道槛儿，只要是过了这个槛的人，会觉得真是太容易了。所有的委托都继承自 MulticastDelegate，编译器在编译时会为委托的声明生成了一个完整的委托类。关于编译器生成的委托类及 Invoke()方法的调用情况，可通过使用 ILDAsm.exe 查看执行文件的 IL 代码获得。

2．动态方法创建

在 C#开发应用中，也可以使用动态方法来创建委托实例。

实例 54：使用动态方法来创建委托实例

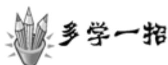
下面通过一个具体的实例来说明使用动态方法来创建委托实例的流程。本实例保存在“光盘：\daima\11”文件夹内，项目名为 usedongtai。实例的功能是根据定义的实例输出对应的文本。实例文件的主要代码如下所示：

```
namespace usedongtai {
    public delegate int weituo(string message);
    public class nn  {
        public int Print(string msg)
        {
            Console.WriteLine(msg);
            return msg.Length;
        }
    }
    public class lei  {
        // 使用委托实例作为参数
        public void chuli(weituo class2)
        {
            class2("==================================");
            class2("        使用委托!!!    ");
            class2("==================================");
        }
    }
    public class Test  {
        public static void Main()
        {
            nn pp = new nn();
            // 创建委托实例
            weituo class2 = new weituo(pp.Print);
            string s = "直接调用委托实例";
            int n = class2(s);       // 直接调用委托实例
            lei pc = new lei();
            pc.chuli(class2);        // 将委托实例作为参数来传递
            Console.ReadKey();
        }
    }
}
```

在上述实例代码中，定义了委托 weituo，并将委托实例 class2 作为参数进行了方法传递，然后通过调用实例方法 chuli()输出了对应的文本。具体执行结果如图 12-2 所示。

```
直接调用委托实例
==================================
        使用委托！！！
==================================
```

图 12-2　实例执行结果



委托跟回调函数是很有渊源的。回调其实跟通知机制有关，考虑这样一个基本的事件序列：a 对象调用了 b 对象的某个方法，希望 b 对象在其方法完成之时调用 a 对象的某个方法。要实现这样的过程，要求 b 对象持有 a 对象的引用（引用一般作为 b 对象方法的参数传入），且“知道”a 对象的那个特定方法。这个方法就是所说的那个回调函数。

3．委托链

因为 C#委托是多路广播的，所以可以将多个委托实例组合在一起，这样就构成了委托链。此时委托链中所有的委托调用列表被链接在一起，组成了一个新的调用列表，这个新列表包含了两个或更多的方法。在 C#中，可以使用二元“+”和“+=”运算符来组合委托，可以使用一元“－－”和“-=”运算符来删除委托实例。

当使用委托组合处理和删除处理后，将会生成一个新的委托。该委托有其独立的调用列表，被组合处理和删除处理后原调用列表将保持不变。

如果在一个委托实例的调用列表内包含多个方法，那么当调用此类委托实例时将会顺序执行调用列表中的各个方法。以上述方式调用的每个方法都会使用相同的参数集。

如果在参数集内包含引用或输出参数，则各个方法的调用都将使用对同一变量的引用。为此，如果调用列表内的某个方法更改了该变量，那么调用列表中排在该方法以后的所有方法所使用的参数都将随之改变。

当委托调用包含引用参数、输出参数或返回一个返回值时，那么委托调用的最后引用或输出参数的值或返回值，就是调用列表中最后一个方法所产生的引用，或输出参数的值或返回值。

12.1.3　使用委托匿名方法

在本章前面介绍的实例中，委托实例的方法都具有相同的名称。C#支持匿名方法的方式，即允许委托的关联代码以内联方式写入使用委托的位置，从而方便直接将代码块绑定到委托实例中。

通过使用匿名方法，可以将代码块直接作为创建委托的参数，而不需要预先定义一个处理方法，并且匿名方法还能够对包含它的函数成员的局部状态进行共享访问。使用委托匿名方法的语法格式如下：

```
delegate (参数表);
  {
      处理代码块
};
```

其中，委托参数列表和返回类型都必须和匿名方法相兼容，这样才能进行从匿名方法到委托类型的隐式转换。

如果一个局部变量或参数的作用域内包含一个匿名方法，那么这个局部变量或参数被称为该匿名方法的外层变量。一个局部变量生存周期仅限于该变量所关联的代码块，或此语句的执行过程中。但是被捕获的外层变量的生存周期将被延长到匿名方法的委托被执行垃圾回

收处理为止。

实例 55：使用匿名方法创建委托

下面通过一个具体的实例来说明使用匿名方法创建委托的具体流程。本实例保存在“光盘：\daima11”文件夹内，项目名为 niming。实例的功能是分别使用命名方法和匿名方法来创建委托。实例文件的主要代码如下：

```
namespace niming {
    public delegate void mm(string s);
    class Test
    {
        static void Main()
        {
            mm p = delegate(string j)     // 用匿名方法来创建委托
            {
                System.Console.WriteLine(j);
            };
            // 调用委托
            p("匿名方法创建的委托。\n");
            p = new mm(Test.chuli);       // 用命名方法来创建委托
           // 调用委托
            p("命名方法创建的委托。");
            Console.ReadKey();
        }
        // 命名方法
        static void chuli(string k)
        {
            System.Console.WriteLine(k);
        }
    }
}
```

上述实例代码的设计流程如下。

1）定义名为 mm 的委托，设置其参数为 s。

2）使用匿名方法来创建委托对象 p。

3）将匿名委托 p 的信息输出。

4）使用关键字 new，创建命名方法的委托 p。

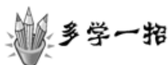
5）命名委托 p 调用方法 chuli()，将对应的信息输出。

上述实例代码执行后，将分别输出显示对应的结果，具体如图 12−3 所示。

匿名方法创建的委托。

命名方法创建的委托。

图 12−3　实例执行结果



在 C#1.1 里，声明和使用委托时要求有委托和一个在委托被触发时具有匹配签名的能够执行的方法，以及一个将命名方法与委托关联的分配语句。匿名方法基本上能够提供与先前命名方法相同的功能，但是它已经不再需要一个在关联到委托之前就明确创建的方法了。

可以把匿名方法想象为一个实现与委托进行关联这项功能的便捷途径。如果同时看一下匿名方法实现和命名方法实现所取得 IL 结果，会发现这两者之间的差别非常小。当编译器碰到匿名方法的时候，它会在类中创建一个命名方法，并将它与委托进行关联。所以匿名方法在运行期间与命名方法的性能非常类似——性能的增加体现在开发人员的生产效率上，而不是运行期间的执行上。

使用匿名方法很简单。只需要把匿名方法放在通常放命名方法的关联语句里。任何在匿名方法里声明的变量的范围都不会超出匿名方法的代码块。但是，匿名方法确实具有访问它们代码块之外的变量的能力，只要这些变量在匿名方法所使用的范围内。这些变量被微软称为外部变量。

12.2　C#事件

事件是类和对象向外发出的信息，用于声明某行为或某处理的条件已经成立。将触发事件的对象称为事件的发送者，将捕获并响应事件的对象称为事件的接收者。在本节的内容中，将详细介绍 C#事件的基本知识。

12.2.1　声明事件

在 C#中使用 event 关键字来声明事件，具体格式如下：

修饰符 even 事件类型 事件名;

在声明事件成员的类中，事件的行为和委托类型的字段很相似。事件存储对某一个委托的引用，此委托表示已经添加到该事件的事件处理方法中。如果没有添加事件的处理方法，则此事件的值为 null。

另外，事件也可以使用访问器的形式来访问，具体格式如下：

```
修饰符 even 事件类型 事件名;{
add
   {
   语句块
   }
remove
   {
   语句块
   }
}
```

C#事件使用修饰符的原则和方法的声明原则相同，事件也分为静态事件、虚事件、密封事件、重写事件和抽象事件。在上述格式内的事件类型必须是委托类型，并且此委托类型必须至少具有和事件本身一样的可访问性。

事件和方法具有相同的签名，签名包括名称和对应的参数列表。事件的签名通过使用委托来定义，例如下面的代码：

```
public delegate void mm(object s,System.EventArgs t);
```

C#事件的主要特点如下。

- ❑ 事件是类用来通知对象需要执行某种操作的方式。
- ❑ 事件一般在图形操作界面中响应用户的操作。
- ❑ 事件通常使用委托来声明。
- ❑ 事件可以调用匿名方法实现。

在.NET 框架的事件签名中，第一个参数通常是触发事件的发送者，第二个参数是第一个传送与事件相关的数据的类。

如果在声明事件时没有采用访问器，则编译器会自动提供访问器。

事件可以作为“+=”运算符左边的操作数，它将被用于将事件处理方法添加到所涉及的事件中，或从事件中删除事件的处理方法。

12.2.2 使用事件

事件的功能是由如下三个关联元素实现的。

1）提供事件数据的类：即 EventNameEventArgs 类，此类从 System.EventArgs 中导出。

2）事件委托：即 EventNameEventHandler。

3）引发事件的类：此类提供事件声明和引发事件的方法。

在现实应用中，通常调用委托来引发事件，并传递与事件相关的参数。委托将调用已经添加到该事件的所有处理方法，如果没有事件处理方法，则该事件为空。

如果要使用在另外一个类中定义的事件，则必须定义和注册一个事件的处理方法。每个事件都可以分配多个处理程序来接收事件。这样事件将自动调用每个接收器，无论接收器有几个，只需调用一次该事件即可引发事件。

在 C#类中实现事件处理的流程如下。

1）定义提供事件数据的类。对 EventNameEventArgs 类进行命名处理，从 System.EventArgs 派生后添加所有事件的成员。

2）声明事件的委托，即对委托 EventNameEventHandler 进行命名处理。

3）使用关键字 event 来定义类中名为 EventName 的公共事件成员，将事件的成员设置为委托类型。

4）在引发事件的类中定义一个受保护的方法。一般是 protected 类型的 virtual 方法。

5）在引发事件的类中确定引发该事件的事件。即调用 OnEventName 来引发该事件，然后使用 EventNameEventArgs 传入事件特定的数据。

如果是在另外一个类中实现事件处理，则具体的实现流程如下。

1）在使用事件的类中定义一个与事件委托有相同签名的事件处理方法。

2）使用对该事件处理方法的一个引用创建委托的实例，当调用此委托实例时会及时自动调用该事件的处理方法。

3）使用“+=”操作符将该委托实例添加到事件。

4）如果不需要事件处理，则使用“-=”操作符将该委托从事件队列中删除。

12.2.3 采用访问器方式使用事件

由本章前面的内容可知，当没有采用访问器方式声明事件时，编译器会自动提供访问器。同样，如果采用访问器方式，也可以实现 12.2.2 节实例中的功能效果。

实例 56：使用 C#事件处理

下面将通过一个实例来说明在 C#中实现事件处理的流程。本实例代码保存在“光盘：\daima\11”文件夹内，项目名为 fangwenqi。本实例的功能是，使用访问器实现订阅处理的功能效果。实例文件的主要代码如下所示：

```
namespace abc
{
    public class mm
    {
        // 定义事件参数类
        public class nn : EventArgs
        {
            public readonly char aa;
            public nn(char aa)
            {
                aa = aa;
            }
        }
        // 定义 delegate
        public delegate void weituo(object sender, nn e);
        // 定义一个私有的委托链字段
        private weituo weituoDelegate;
        // 用 event 关键字声明事件对象
        public event weituo TestEvent
        {
            // 将方法添加到委托链中
            add
            {
                weituoDelegate += value;
            }
            // 从委托链中删除方法
            remove
            {
                weituoDelegate -= value;
            }
        }
```

```
        // 事件触发方法
        protected virtual void OnTestEvent(nn e)
        {
            if (weituoDelegate != null)
                weituoDelegate(this, e);
        }
        // 引发方法
        public void RaiseEvent(char aa)
        {
            nn e = new nn(aa);
            OnTestEvent(e);
        }
    }
    // 侦听事件的类
    public class zz
    {
        // 定义处理事件的方法，它与声明事件的 delegate 具有相同的参数和返回值类型
        public void KeyPressed(object sender, mm.nn e)
        {
            Console.WriteLine("发送者为：{0},所按的键为：{1}",sender, e.aa);
        }
        // 订阅事件
        public void Subscribe(mm eventSource)
        {
            eventSource.TestEvent += new mm.weituo(KeyPressed);
        }
        // 取消订阅事件
        public void UnSubscribe(mm eventSource)
        {
            eventSource.TestEvent -= new mm.weituo(KeyPressed);
        }
    }
    // 测试类
    public class Test
    {
        public static void Main()
        {
            // 创建事件源对象
            mm es = new mm();
            // 创建侦听对象
            zz el = new zz();
            // 订阅事件
            Console.WriteLine("开始订阅事件\n");
            el.Subscribe(es);
            // 引发事件
            Console.Write("输入一个字符，再按 Enter 键：");
```

```
            string s = Console.ReadLine();
            es.RaiseEvent(s.ToCharArray()[0]);
            // 取消订阅事件
            Console.WriteLine("\n 开始取消订阅事件\n");
            el.UnSubscribe(es);
            // 引发事件
            Console.Write("输入一个字符，再按 Enter 键：");
            s = Console.ReadLine();
            es.RaiseEvent(s.ToCharArray()[0]);
            Console.ReadKey();
        }
    }
}
```

上述实例代码的设计流程如下。

1）定义类 mm 和 nn，并设置 nn 的事件参数类。

2）定义委托 weituo，并声明事件的对象 TestEvent。

3）定义事件触发方法 OnTestEvent()，如果事件不为空则引发这个方法。

4）定义侦听事件的类 zz。

5）声明事件处理方法 KeyPressed()，显示对应的提示文本。

6）分别通过方法 Subscribe()和 UnSubscribe()实现事件激活功能和事件取消功能。

7）分别创建事件发送者对象 es、侦听对象 el。

8）分别调用不同事件状态下的处理方法将对应的提示文本输出显示。

执行后将首先显示预设的提示文本，如图 12-4 所示；当用户按某按键后，会激活对应的事件并调用对应的处理方法，输出显示用户的按键名，并显示取消事件的提示文本，如图 12-5 所示。当再次按某按键后会删除对该事件的订阅。

```
开始订阅事件

输入一个字符，再按Enter键：
```

图 12-4　实例执行结果

```
开始订阅事件

输入一个字符，再按Enter键：g
发送者为：fangwenqi.mm，所按的键为：

开始取消订阅事件

输入一个字符，再按Enter键：h
```

图 12-5　实例执行结果

12.3　C #4.0 新特性——动态编程

C# 4.0 的主题就是动态编程(Dynamic Programming)。虽然 C#仍然是一种静态语言，但是对象的意义开始变得越来越“动态”。它们的结构和行为无法通过静态类型来捕获，或者至少编译器在编译程序时无法得知对象的结构和行为。

12.3.1　再议 dynamic

从 C#4.0 开始，引入了一种新的静态类型“dynamic”，当拥有了一个 dynamic 类型的对

象后，只会在运行时解析“对它做的事情”。假设有两个分别表示两种汽车的类：

```
public class Car
{
public string GetName()
{
return "You selected Red Car.";
}
}
public class Bus
{
public string GetName()
{
return "You selected Red Bus.";
}
}
```

现在，就可以用 dynamic 类型来表示这两种汽车。写一个函数 GetCar()，根据用户不同的选择返回不同的对象。

```
static private Object GetLei(int i)
{
if (i == 1)
{
return new Bus();
}
else  // default
{
return new Car();
}
}
static void Main(string[] args)
{
Console.WriteLine("Please Select Your Lei: 1 -- Bus; 2 -- Car");
int nLeiType = Console.Read();
dynamic Lei = GetLei( nLeiType );
Console.WriteLine( Lei.GetName() );
}
```

C#编译器允许编译者通过 dynamic 对象调用任何方法，即使这个方法根本不存在，编译器也不会在编译的时候报编译错误。只有在运行的时候，它才会检查这个对象的实际类型，并检查在它上面 GetName()是什么意思。动态类型将使得 C#可以以更加统一而便利的形式表示下列对象。

- 来自动态编程语言——如 Python 或 Ruby 的对象。
- 通过 IDispatch 访问的 COM 对象。
- 通过反射访问的一般.NET 类型。

❑ 结构发生过变化的对象——如 HTML DOM 对象。

当得到一个动态类型的对象时，不管它是来自 COM 还是 IronPython、HTML DOM 还是反射，只需要对其进行操作即可，动态语言运行时(DLR)会帮助指出针对特定的对象以及这些操作的具体意义。这将给开发带来极大的灵活性，并且能够极大程度上地精简代码。

12.3.2 动态编程的应用

在下面的内容中，以一个具体实例来演示 C# 4.0 中的动态编程是如何简化对 Silverlight 应用程序中 HTML DOM 对象的访问的。

（1）安装 Silverlight 2 runtime 和 SDK

为了运行这个实例，需要安装 Silverlight 2 runtime 和 SDK。这些都可以从微软的网站上直接下载：

❑ Silverlight 2 RTW runtime。

❑ Silverlight 2 RTW SDK。

另外，因为 Visual Studio 2010 跟 Silverlight 的一些已知的兼容性问题，还需要一个拥有管理员权限的账号来运行 Visual Studio 2010 CTP。

（2）下载实例项目 SilverlightSolution

为了演示 C# 4.0 的诸多新特性，微软通过 MSDN Code Gallery 发布了一个演示 C#新特性的实例程序包，所用到的实例项目 SilverlightSolution 就在这个程序包中。可以访问：

http://code.msdn.microsoft.com/csharpfuture

下载此实例程序包后，将需要的项目 SilverlightSolution 解压到 C 盘根目录下。

（3）编译并运行实例项目

用管理员身份启动 Visual Studio CTP，然后打开 SilverlightSolution 项目，将得到一个空的 Silverlight 程序页面，如图 12−6 所示。

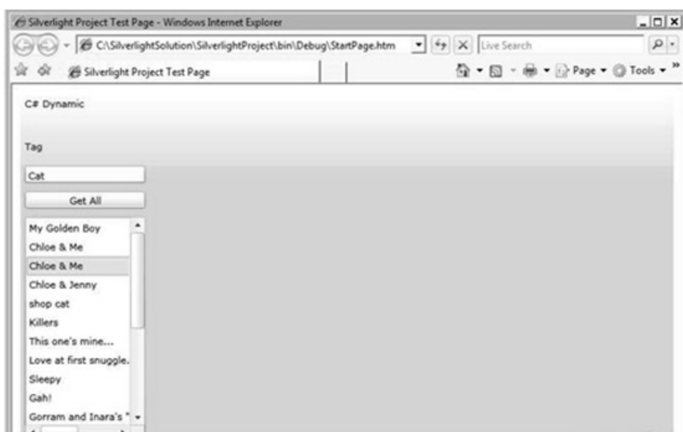


图 12−6　Silverlight 程序界面

（4）添加对 HTML DOM 对象的动态访问

在 Page 类中添加两个 dynamic 类型的对象 doc 和 win，分别用于表示当前页面和窗口，

修改后的 Page 类如下：

```
public partial class Page : UserControl{
// 添加的代码
dynamic doc = HtmlPage.Document.AsDynamic();
dynamic win = HtmlPage.Window.AsDynamic();
//…
}
```

然后，修改 Page 类的函数 btnSearch_Click()，当用户单击”Get All”按钮后，更新用户搜索的内容，修改窗口的标题。

```
void btnSearch_Click(object sender, RoutedEventArgs e){
doc.Title = "Pictures of " + txtTag.Text;
lstPictures.ItemsSource = Source.LoadItems();
}
```

（5）添加 Virtual Earth Jscript 控件

打开解决方案中的测试页 StartPage.htm，在<title>标签的前面添加如下 Jave Script 代码，引用 Virtual Earth 控件：

```
<script type="text/javascript" src="http://dev.virtualearth.net/mapcontrol/
mapcontrol.ashx?v=6.1"/>
```

然后，在<body>标签中添加如下代码，定义地图控件的位置和大小。只需通过简单的两步，就完成了对测试页面的修改。接下来即可通过 C#访问和控制 Virtual Earth 控件了。

（6）用 C#代码控制 Virtual Earth 控件

在 Page 类中添加如下代码控制 Virtual Earth 控件，以实现获取地图，添加 Pin 的功能。代码如下：

```
dynamic map = null;
       void GetMap()
{
map = win.New.VEMap("myMap");
map.LoadMap();
}
       void AddPin(Item item)
{
dynamic loc = win.New.VELatLong(item.Latitude, item.Longitude);
var pin = map.AddPushpin(loc);
pin.SetTitle(item.Title);
pin.SetDescription(item.Description);
map.SetCenterAndZoom(loc, 7);
}
```

为了体会动态编程所带来的便利，接下来看看在没有动态类型的 C# 3.0 中，AddPin()函数

应该如何实现。如果是在 C# 3.0 中，doc，win 和 map 都将是静态类型，为了执行这些对象的某些方法，需要进行显式的类型转换，同时，还需要将方法名作为字符串传递以实现方法的调用。很明显，C# 3.0 中的实现过程非常繁琐，而 C# 4.0 中的实现是如此的简单、自然：

```
void AddPin(Item item){
ScriptObject loc = win.CreateInstance("VELatLong", item.Latitude,
item. Longitude);
ScriptObject pin = (ScriptObject)map.Invoke("AddPushpin", loc);
pin.Invoke("SetTitle", item.Title);
pin.Invoke("SetDescription", item.Description);
map.Invoke("SetCenterAndZoom", loc, 7);
}
```

接下来需要在适当的位置调用这两个函数，以实现地图的加载和 Pin 的添加。在 Page 类的 Init()函数中添加 GetMap()函数调用：

```
public void Init(){
GetMap();
}
```

另外，在 lstPictures_SelectionChanged()函数中添加 AddPin()函数，当用户选择的图片变化后，重新获取 Pin 的位置并添加到地图上：

```
void lstPictures_SelectionChanged(object sender,SelectionChangedEventArgs e){
Item selected = lstPictures.SelectedItem as Item;
AddPin(selected);
}
```

（7）编译解决方案

至此，对项目的修改就全部完成了。重新编译整个解决方案，可以看到添加 Virtual Earth 控件后的页面。当在左侧列表中选择图片后，Pin 会定位到这幅图片拍摄的地点，当把鼠标移动到 Pin 上时，则会显示图片和更多的相关信息，如图 12-7 所示。

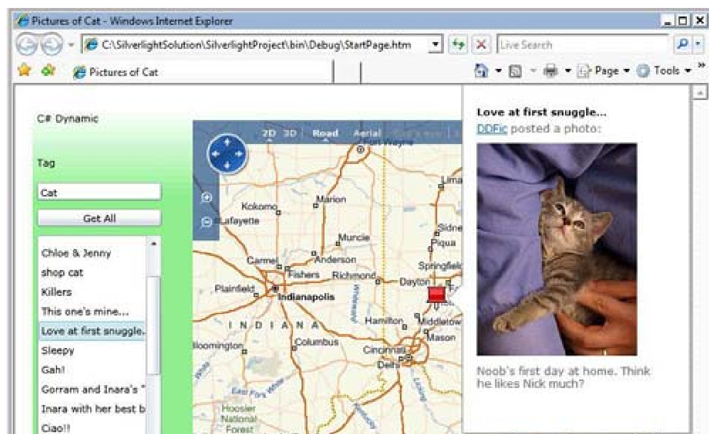


图 12-7　添加 Virtual Earth 后的页面

12.4 疑难问题解析

本章详细介绍了 C#中委托和事件的基本知识。本节将对本章中比较难以理解的问题进行讲解。

读者疑问：C#委托有什么特点？

解答：C#委托的特点如下。

- ❑ 委托类似于 C++ 函数指针，但它是类型安全的。
- ❑ 委托允许将方法作为参数进行传递。
- ❑ 委托可用于定义回调方法。
- ❑ 委托可以链接在一起。例如，可以对一个事件调用多个方法。
- ❑ 方法不需要与委托签名精确匹配。
- ❑ 从 C# 2.0 开始，引入了匿名方法的概念，此类方法允许将代码块作为参数传递，以代替单独定义的方法。



职场点拨——跳槽的成本

都说铁打的营盘流水的兵，可能某些人会觉得当前所在的岗位有些屈才，可能某些人会对当前待遇不满，也可能会觉得工作压力大。人往高处走，水往低处流，作为职场中的程序员，肯定会在某一个时期产生寻找更好的工作的想法，这很容易理解。但是在决定之前，应该先考虑一下跳槽的成本。

跳槽所产生的成本可以分为两种，分别是显性成本和隐性成本。显性成本就是可以明确预见的成本。在此假设程序员 mm 先生想跳槽，其显性成本为：原单位的奖金损失 20000 元；年内的各项福利损失约 15000 元；对新工作有针对性的“充电”费用 8000 元；在新单位两年的服务期未满，单方面提出解除劳动合同，须支付一大笔违约金 75000 元(100000 元÷24 个月×18 个月)，mm 先生此次跳槽的显性成本总共 118000 元。

隐性成本则是不容易发现的，例如程序员 mm 想跳槽，下家公司离家较远，就可能会租住离公司较近的房子或购置车辆等交通工具；新公司对着装严格，就需添置新的工作装；初到新公司，对人员不熟悉，要与新同事以搞好关系需要花费人际交往费；接触新的工作，对各种规定不熟悉，加班加点到深夜以便跟上同事的步伐等，这些都是需要花大量时间、精力、金钱的。因此跳槽也许在工资上能够带来一定的提高，但是由于不断更换环境而有着更大的花费。以程序员 mm 为例，他这次跳槽所花费的隐性成本约为 20000 元。

这样的话，即使 mm 先生此次跳槽比较风光，工资比原先高出 50%。但实际上综合所有的跳槽成本后，实际却损失了 100500 元(新公司工作 9 个月收入 112500 元−如在原公司工作 9 个月收入 75000 元−跳槽的显性以及隐性成本 138000 元)。

第13章 泛　　型

自从 C#2.0 开始，便引入了泛型这一概念。泛型的类、结构、接口、委托和方法，可以根据对应的存储和操作的类型实现参数化处理。使用泛型后，可以实现更强的编译时的类型检查，减少数据类型间的显示转换和装箱操作时的类型检查。介绍完本章的 C#泛型知识后，宣告整个 C#语言的基本语法部分介绍完毕。希望读者仔细阅读本书的知识，对各个知识点做到融会贯通。通过本章能学到如下知识。

- 泛型声明。
- 泛型约束。
- 泛型集合类。
- 泛型迭代器。
- 职场点拨——认清你的职责。

2008 年 X 月 X 日，多云

今天的工作还算轻松，当我刚刚完成项目经理安排给我的紧急程序任务时，同事央求我帮她做图片处理。我向来喜欢 PS 处理，并且闲暇时间经常处理自己的户外照片，所以就答应了她的请求。当然我还有一个私心，自己干好本职工作的同时，帮助了同事，为公司多干了活，也许会得到领导的器重的。



一问一答

小菜：我今天发挥了一下，帮助同事处理了几张图片，她夸我是全才！

Wisdom：还不错，但是你的本职工作做好了吗？

小菜：当然做好了，不然我也不会去帮忙的。我想我给公司多干了活，应该会得到领导的器重的。

Wisdom：嗯，在完成本职工作的前提下可以帮助你的同事。在本章的最后，我详细讲解了认清自己职责的问题，你可以仔细体会其中的真谛！

小菜：好的，言归正传，本章讲的泛型有什么作用？

Wisdom：C#是一种创新的语言，为了能够更好地解决项目中的问题，推出了泛型这一概念。

13.1　泛型概述

C#泛型是一种编程方法，是处理算法、数据结构和其他软件概念的抽象表示。推出泛型

的目的是，采用广泛适用和可交互性的形式来表示算法和数据结构，以使它们能够直接用于软件的构造。通过泛型，可以由一个定义的类派生出许多特定的类型，这样可以节约开发人员的开发时间和精力。

13.1.1　泛型的特点

泛型具有如下四个特点。

1）适用数据抽象的最小假设来表示算法，并用算法的最小假设来表示数据抽象，来提高他们之间的交互性。

2）在不降低效率的前提下，可以最大限度地将具体算法转化为一般化算法。

3）当提升结果的一般性不能够满足算法的所有应用时，则另外提供一种更为普通的形式。

4）为同一个目的提供多种泛型算法，并确保它们具有相同的抽象性。

C#泛型借鉴了 C++模板技术，通过使用泛型后，创建一个集合将变得十分简单，但是往往会降低程序的可读性。

13.1.2　泛型的优点

如果不使用泛型，在 C#中可以使用 object 类型作为通用的数据结构，以存储任何类型的数据。看下面的一段代码：

```
public class class1;{
  object [] items;
  int aa,
  int bb,
  int cc;
  public void chuli(object item)
          {
          ……
          }
public object F()
          {
          ……
          }
}
```

在上述代码中，定义类 class1 将数据存储在一个 object 数组中。类 mm 的方法 chuli()和 F()分别使用 object 来接收和返回数据。

在上述代码中，能够将任何类型的值传送到队列内。如果再在列内检索到某个值时，必须将 F()方法的结果显式的强制转换回相对应的类型。这样会加大代码的编写量，并且在运行时会造成额外的开销。上述过程的实现代码如下：

```
class1 class11=new class1 ();
class11.chuli(new F1());
F1 c=(F1)mm1.F();
```

另外，当将一个值类型的值传递给方法 chuli()时，此值会自动被装箱。当以后再次检索该值时，必须使用显式类型强制转换来取消这个装箱。例如，如果值类型的值是 2，则需要使用下面的代码进行处理：

```
mm mm1=new mm();
mm1.chuli(2);                          //装箱处理
int I = (int) mm1.F();                 //拆箱处理
```

通过上述装箱和拆箱处理后，会涉及动态内存分配和运行时类型检查，这样就额外的增加了性能开销，降低了效率。并且上面的类 class1 无法对队列上的数据类型进行限制。这样，如果将一个 F1 实例送入到队列后，却在检索到该实例之后意外的将它强制转换为错误类型。例如，适用下面的代码：

```
class1 class11=new class1 ();
class11.chuli(new F1());
string ss =(string) class11.F();
```

在上述代码中，错误的使用了 class1 对象，因为出队对象类型是 F1 对象，不能向 string 类型转换，代码执行后会出现异常错误。

13.2　泛型声明

通过类、结构、接口、委托和方法都可以声明泛型。声明泛型的结构和声明泛型类的格式基本类似。为此，在本节的内容中，将着重介绍类、接口、委托和方法的泛型声明。

13.2.1　声明泛型类

声明 C#泛型类的格式如下：

```
类修饰符 class 类名 <类型形参表> ：基类\接口
{
类体
}
```

其中，类型形参表是泛型类型或方法定义中的占位符列表。虽然泛型声明可以嵌套在非泛型类声明中，但是任何嵌套在泛型类声明或泛型结构声明中的类本身就是一个泛型类声明。

每个类型的形参都是一个简单的标识符，代表一个为创建构造类型而提供的类型实参的占位符。类型形参是将来提供的类型的形式占位符。例如，下面的代码就是一个泛型队列类的声明。

```
public class mm<T>;{
  T [] items;
  int aa,
  int bb,
```

```
        int cc;
        public void chuli(T item)
            {
            ……
            }
        public T F()
            {
            ……
            }
    }
```

在上述代码中，设置了 T 是类型的形参，队列元素的类型为 T，然后通过方法 chuli()将 T 类型元素传入到队列，通过 F 从队列中取出 T 类型的元素。

在声明泛型类型时，它本身表示未绑定的泛型类型，可以生成多种不同类型的“模板”。类型实参被放在泛型类型声明名称后的尖括号对“< >”之间，在创建构造类型时能够替换类型形参的实际参数。当使用上面的泛型类 mm<T>时，能够指定用于替换 T 的实际类型。例如在下面的代码中，指定了 int 为 T 的类型实参。

```
    mm<int> mm1 =new mm<int>();
        mm1.chuli(3);
        int aa=mm1.F(),
```

在上面的代码中，mm<int>就是泛型类 mm<T>的构造类型。在 mm<int>类型中，出现的每一个 T 都被替换为类型实参 int。当创建 mm<int>的实例后，items 数组的实际类型是 int[]而不是 object[]。这样，就比非泛型处理提供了更高的效率。同样，mm<int>的方法 chuli()和 F 所操作的也是 int 类型的值。如果将其他类型的值放入到队列，会产生编译错误。

泛型提供了强类型机制，即如果将一个 int 值放入到 F1 对象的队列，将会导致错误。

在声明泛型类型时，可以包含任意数目的类型参数。虽然上面代码中的 mm<T>只有一个类型形参，但是在下面代码中，泛型 nn 类具有两个类型的形参，一个用于键类型，一个用于值类型。

```
    public class nn<A,B>;
    {
        public void Add(A key,B value);{…}
        public B this[A key]; {…}
    }
```

当使用上面的 nn<A,B>类型时，必须提供两个类型实参。例如下面的代码：

```
    Dictionary<string, F1> vv = new Dictionary<string, F1>();
    vv.Add("厉害",new F1());
    F1 c = vv["厉害"];
```

如果是值类型，在每次泛型类型实例化和运行.NET 公共语言时，都会创建单独的本机代码。如果是引用类型，则会共享该本机代码。

在类声明中，指定的基类型或基接口可以是构造类类型，但不能是类型形参，但是在基类或基接口的作用域中可以包含类型形参。看下面的一段代码：

```
class c<uint,v>{
            ...
        }
        interface mm<v>{
            ...
        }
        class d:c<string,int>,mm<string>{
            ...
        }
        class e<t> : checked<int,t>,mm<t>{
            ...
        }
        class nn<v> : v                     //出现错误
         {
            ...
        }
```

在上述代码中，类 nn 不能使用基类 v 作为类型形参。

可以直接把泛型的所有接口成员作为构造类型的一部分，作为任何包容类中的类型形参。当公共运行语言在使用特定的封闭构造类型时，所出现的所有形参都将会被替换，被替换为该构造函数类型提供的实际类型实参。在实例函数中，this 是包含这些成员声明的实例类型。

13.2.2　泛型接口

声明泛型接口的格式如下：

```
修饰符  interface 接口名 <类型形参表> : 基接口
{
   接口体
}
```

从上述格式可以看出，在声明泛型接口时，只是比声明普通接口的格式增加了“类型形参表”。

在声明泛型类型时，实现的接口必须和所有的构造函数保持唯一，否则将不能确定该为那些构造类型调用哪个方法。例如，通过如下格式可以声明一个泛型类型：

```
interface a <T>
{
   void F();
}
class mm <x,y> : a<x>,a<y>
{
```

```
        void a<x>.F();
              {
                     ……
              }
        void a<y>.F();
              {
                     ……
              }
    }
```

如果使用下面的代码执行构造函数：

```
    a<int> x =new X<int,int> ();
    x.F();
```

类 x 就不能确定调用的到底是哪一个方法。

在 C#泛型中，可以统一处理不同继承级别指定的接口。看下面的一段代码：

```
    interface a <T>
    {
        void F();
    }
    class mm <D> : a<D>
    {
        void a<D>.F();
              {
                     ……
              }
    }
    class nn <D,Q> : mm<D>,a<Q>
    {
        void a<Q>.F();
          {
               ……
          }
    }
```

在上述代码中，虽然“nn <D,Q>”同时实现了 a<D>和 a<Q>，但是上述代码是有效的。使用下面的代码可以调用类 nn 中的方法 F（）：

```
    a<int> x =new X<int,int> ();
    x.F();
```

在上述代码中，nn<int,int>实际上重现了 a<int>。

13.2.3 泛型委托

声明 C#泛型委托的格式如下：

```
修饰符  delegate 返回类型 委托名 <类型形参表> (形参表);
```

从上述格式可以看出，声明泛型委托只是比声明普通委托的格式增加了“类型形参表”。

在声明泛型委托时，必须提供类型实参后才能创建构造委托类型。将委托声明中的每个类型行参，替换为构造委托类型的对应类型实参，这样就实现了构造委托的形参类型和返回类型的创建。返回的结果类型和形参类型，能够确定哪些方法与构造委托类型兼容。例如下面的代码：

```
delegate bool mm <T> (T value);
class a{
   static bool F(int i){…}
   static bool G(string s ){…};
   static void Main()
            {
              mm<int> mm1=F;
              mm<string> mm2=G;
            }
}
```

在上述代码中，两个赋值等效于下面格式的代码：

```
static void Main() {
              mm<int> mm1=new mm<int>(F);
              mm<string> mm2= new mm<string>(G);
        }
```

13.2.4　泛型方法

泛型的方法包括了类型形参，它可以在类、结构或接口的声明中进行声明，这里的类、接口或结构既可以是泛型，也可以不是泛型。如果在泛型类型中声明泛型方法，这个方法体可以引用该方法类型的形参，也能引用包含该方法声明的类型形参。

声明 C#泛型方法的格式如下：

```
修饰符 返回类型 方法名 <类型形参表> (形式参表);
{
     方法体
}
```

从上述格式可以看出，在声明泛型方法时，只是比声明普通方法的格式增加了“类型形参表”。泛型的方法有如下两个形参表。

1）泛型类型形参表。

2）形参列表。

类型参数可以作为返回类型或形参的类型出现，方法类型的形参名不能和当前方法内的普通形参名相同。例如下面的代码定义了泛型方法，并实现对泛型方法的调用：

```
static void mm<T>(ref T a,ref T b) {
```

```
                T tt;;
                Tt=a;
                a=b;
                b=tt;
            }
    public static void Diaoyong() {
                int x=1;
                int y=3;
                //开始调用
                mm<int>(ref x,ref y);
                System.Console.WriteLine(x+””+b);
                Console.ReadKey();
            }
```

泛型方法的签名还包括泛型类型形参的数目和类型行参的序号位置。另外，泛型方法的隐藏、重载和重写规则和非泛型方法相同，并且泛型方法也可以是虚的、抽象的或封闭的。泛型的方法可以使用许多类型参数进行重载。

看下面的一段代码，代码中的方法可以全部放在同一个类中：

```
    void mm(){}
    void mm<T>(){}
    void mm<T,Q>(){}
```

13.3 泛型约束

在定义泛型类时，可以限制在实例化类客户端代码的参数类型。当客户端代码使用了被约束的类型来实例化时，会出现编译错误。在本节的内容中，将详细介绍 C#泛型约束的基本知识。

在 C#中，使用关键字 where 来实现约束。

1. 结构约束

结构约束的类型参数必须是值类型，它几乎可以指定除 Nullable 以外的所有值类型。结构约束的具体格式如下：

```
    T(类型):结构
```

2. 类约束

类约束的类型参数必须是引用类型，包括所有的类、接口、委托和数组类型。类约束的具体格式如下：

```
    T(类型):类
```

3. 新建约束

新建约束的类型参数必须是具有无参数的公共构造函数，当此约束类型和其他约束类型一起使用时，必须在最后使用新建约束。新建约束的具体格式如下：

```
T(类型):new()
```

4. 基类约束

基类约束的类型参数必须是指定基类或是派生指定的基类。基类约束的基本格式如下：

```
T(类型):<基类名>
```

5. 接口约束

接口约束的类型参数必须是指定的接口，或者是正在实现过程中的指定接口。接口约束的具体格式如下：

```
T(类型):<接口名>
```

6. 裸类约束

通过裸类约束，为 T 提供的参数必须是为被约束对象提供的参数，或者是派生为被约束对象提供的参数，这被称为裸类约束。裸类约束的具体格式如下所示：

```
T(类型):U(被约束对象)
```

在某程度上，编译器必须保证它调用的运算符或调用的方法，能够操作客户端代码指定的类型参数，这样就能够检查泛型列表中某个项，以确定这个项是否有效，或和它的其他项进行比较。

实例 57：使用泛型类型约束的方法

下面通过一个具体的实例来说明使用泛型类型约束的具体流程。本实例保存在“光盘：\daima\13”文件夹内，项目名为 fanyueshu。实例的功能是根据定义的泛型类，调用类内的方法并对其进行处理，并输出处理后的文本。实例文件的主要代码如下：

```
namespace fanyueshu {
    public class mm
    {
        private string name;
        public string Name
        {
            get
            {
                return name;
            }
            set
            {
                name = value;
            }
        }
        public mm(string s)
        {
```

```
            name = s;
        }
    }
    public class nn<T> where T : mm
    {
        ArrayList names = new ArrayList();
        public void Add(T t)
        {
            names.Add(t.Name);
        }
        public void shuchu()
        {
            foreach (string s in names)
            {
                Console.WriteLine(s);
            }
        }
    }
    public class Program
    {
        static void Main()
        {
            nn<mm> tt = new nn<mm>();
            tt.Add(new mm("AA"));
            tt.Add(new mm("BB"));
            tt.Add(new mm("CC"));
            tt.Add(new mm("DD"));
            tt.Add(new mm("EE"));
            tt.Add(new mm("FF"));
            Console.WriteLine("世界杯 6 强");
            tt.shuchu();
            Console.ReadKey();
        }
    }
}
```

上述实例代码的设计流程如下。

1）定义类 mm。

2）在类 mm 内设置属性 name，并返回 name 值。

3）定义方法 shuchu()，设置其参数 s 为 string 类型。

4）定义泛型类 nn，设置其泛型类型为 T。

5）定义数组 names，并定义方法 Add()向数组内添加数据。

6）通过 foreach 语句逐一读取添加后的数组数据，并将数据输出。

7）定义 mm 类型对象 tt，然后调用输出函数输出显示处理后的数据。

13.4　泛型集合类

泛型被推出后，主要被应用于泛型集合。在.NET 4.0 类库中，提供了命名空间 System.Collections.Generic，在其中定义了常用于泛型的集合类。通过这个泛型集合类，不但可以提高类型的安全性，而且能够更好地运行。在本节的内容中，将简要介绍 C#泛型集合类的基本知识，并通过对应实例的实现来讲解泛型集合类的具体使用流程。

13.4.1　泛型集合类介绍

在 C#中，有如下几种最为常用的泛型集合类。

- List<T>，对应于非泛型集合类的 ArrayList。
- Dictionary<TKey,TValue>，对应于非泛型集合类的 Hashtable。
- Queue<T>，对应于非泛型集合类的 Queue。
- Stack<T>，对应于非泛型集合类的 Stack。
- StoreList<T>，对应于非泛型集合类的 StoreList。
- LinkList，没有对应的非泛型集合类，功能是提供运算复杂度为 O(1)的插入和删除操作。
- StortedDictionaryt，没有对应的非泛型集合类，是一个排序字典，其插入和检索操作的运算复杂度为 O(log n)，是 StoreList 的游泳替代类型。
- KeyedCollection，介于列表和字典之间的复合类型，功能是提供了一种存储包含自己键的对象方法。

在实际项目开发时，建议读者多使用泛型集合类，这样不但可以获得更好的类型安全，而且不需要从基集合类派生并实现类型特定的成员。

C#中的泛型集合和对应的非泛型集合的用法类似，但是泛型类型集合能够提供非泛型集合所没有的功能。例如，List<T>可以接受泛型的委托方法，并允许指定的、用于排序和搜索列表的 IComparer 泛型接口实现。

13.4.2　泛型集合类的使用

泛型集合类在现实的开发应用中得到广泛的使用，这是因为泛型集合的功能十分强大，并且能够提高类的安全性和提高程序运行效率。在下面的内容中，将详细介绍常用泛型集合类的使用方法。

1．基本处理

这里的基本处理是指添加和删除存储列表的数据等常规操作。基本处理的流程如下。

1）设置一个容量为 0 的字符串列表。

2）通过显示容量属性。

3）使用 Add（）方法进行添加处理，使用 Remove 来删除指定的数据。

实例 58：使用泛型集合类

下面通过一个实例来说明使用泛型集合类的基本流程。本实例保存在“光盘：\daima\13”文件夹内，项目名为 fanjihe。实例的功能是预先定义字符串列表，然后调用集合类的操

作方法和属性，并操作处理列表内的数据。实例文件的主要代码如下所示：

```
namespace fanjihe
{
    public class Test
    {
        public static void Main()
        {
            List<string> mm = new List<string>();

            Console.WriteLine("现在的出线球队: {0}", mm.Capacity);

            mm.Add("AA");
            mm.Add("BB");
            mm.Add("CC");
            mm.Add("DD");

            Console.WriteLine();
            foreach (string club in mm)
            {
                Console.WriteLine(club);
            }

            Console.WriteLine("\n现在的出线球队: {0}", mm.Capacity);
            Console.WriteLine("数量: {0}", mm.Count);

            Console.WriteLine("\n包括了AA: {0}",
                mm.Contains("AA"));

            Console.WriteLine("\n添加EE后的球队有：");
            mm.Insert(2, "EE");

            Console.WriteLine();
            foreach (string club in mm)
            {
                Console.WriteLine(club);
            }

            Console.WriteLine("\n现在的球队：mm[3]: {0}", mm[3]);

            Console.WriteLine("\n容量: {0}", mm.Capacity);
            Console.WriteLine("数量: {0}", mm.Count);

            Console.WriteLine("删除ＡＡ");
            mm.Remove("ＡＡ");

            Console.WriteLine();
```

```
                foreach (string club in mm)
                {
                    Console.WriteLine(club);
                }

                mm.TrimExcess();
                Console.WriteLine("\n使用TrimExcess()处理后");
                Console.WriteLine("容量: {0}", mm.Capacity);
                Console.WriteLine("数量: {0}", mm.Count);

                mm.Clear();
                Console.WriteLine("\n使用Clear()处理后");
                Console.WriteLine("容量: {0}", mm.Capacity);
                Console.WriteLine("数量: {0}", mm.Count);
                Console.ReadKey();
            }
        }
    }
```

上述实例代码的设计流程如下。

1）定义集合类对象 mm。

2）设置 mm 的初始值为 0，并输出对应结果的文本。

3）通过 Add()方法向列表内添减 4 个数据。

4）通过 foreach 读取并输出显示添加后的列表数据。

5）通过 Count()方法获取并显示当前列表内的数据数目。

6）判断列表 mm 内是否已包含“AA”。

7）使用 Insert()方法添加“EE”。

8）foreach 读取并输出显示添加后的列表数据。

9）通过 Remove()方法删除列表对象 mm 内的数据“AA”。

10）分别调用 TrimExcess 和 Clear，将列表 mm 的数据输出并显示。

执行后将使用编写的操作方法操作处理列表对象 mm 的数据，并输出显示处理后的对应数据，如图 13-1 所示。

```
AA
BB
EE
CC
DD

现在的球队：mm[3]: CC

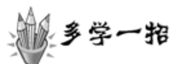
容量: 8
数量: 5
删除AAA

BB
EE
CC
DD

使用TrimExcess()处理后
容量: 4
数量: 4

使用Clear()处理后
容量: 4
数量: 0
```

图 13-1　实例执行结果



在实际项目开发时，建议读者使用泛型集合类，这样可以获得更好的类型安全，并且不需要从基集合类派生并实现类型特定的成员。C#中泛型集合和对应的非泛型集合的具体使用类似，但是泛型类型集合能够提供非泛型集合所没有的功能。例如，List<T>可以接受泛型的委托方法，并允许指定的、用于排序和搜索列表的 IComparer 泛型接口实现。

2．键值处理

键值处理可以对有字符串键的对象进行添加和删除处理操作。键值处理的基本流程如下。

1）预先设置一个具有 Dictionary<TKey,TValue>的对象。

2）使用 Add()方法进行添加处理。

3）使用 Remove 删除指定的数据。



当添加重复的键值时会引发异常错误。

实例 59：使用泛型集合类的键值

下面通过一个具体的实例来说明使用泛型集合类的键值处理流程。本实例保存在“光盘：\daima\13”文件夹内，项目名为 usejianjihe。实例的功能是预先定义键值对象，然后调用集合类的操作方法和属性，对对象内的数据进行操作处理。实例文件的主要代码如下：

```
namespace usejianjihe {
    public class Test {
        public static void Main()
        {
            Dictionary<string, string> mm =
                new Dictionary<string, string>();
            mm.Add("aaa", "a.exe");
            mm.Add("bbb", "b.exe");
            mm.Add("ccc", "b.exe");
            mm.Add("ddd", "d.exe");
            // 如果试图添加一个已经存在的键，Add（）方法会显示异常
            try
            {
                mm.Add("aaa", "aa.exe");
            }
            catch (ArgumentException)
            {
                Console.WriteLine("有一个元素已经使用了键 = \"aaa\"。");
            }
            Console.WriteLine("对应键 \"ddd\" 的值为 {0}。",
                mm["ddd"]);
            // 使用索引器来赋值时，如果键已经存在，那么就会修改对应的值，而不会引发异常
            mm["ddd"] = "dd.exe";
            Console.WriteLine("对应键 \"rtf\" 的值为 {0}。",
                mm["ddd"]);
            // 使用索引器时，如果键不存在，就会添加一个元素
            mm["ggg"] = "dd.exe";
            // 使用索引器来获取值时，如果键不存在，就会引发异常
            try
            {
                Console.WriteLine("对应键 \"eee\" 的值为 {0}。",
```

```
            mm["eee"]);
}
catch (KeyNotFoundException)
{
    Console.WriteLine("没有找到对应键 \"eee\" 的值。");
}
// 当程序必须经常尝试获取字典中不存在的键值时，可以使用 TryGetValue（）方法
// 作为一种更有效的检索值的方法
string value = "";
if (mm.TryGetValue("eee", out value))
{
    Console.WriteLine("对应键 \"eee\" 的值为 {0}。", value);
}
else
{
    Console.WriteLine("没有找到对应键 \"eee\" 的值。");
}
// 插入元素之前，可以先用 ContainsKey 来判断键是否已经存在
if (!mm.ContainsKey("fff"))
{
    mm.Add("fff", "hypertrm.exe");
    Console.WriteLine("已经添加了对应键 \"fff\" 的值: {0}",
        mm["fff"]);
}
Console.WriteLine();
foreach (KeyValuePair<string, string> kvp in mm)
{
    Console.WriteLine("Key = {0}, Value = {1}",
        kvp.Key, kvp.Value);
}
Dictionary<string,string>.ValueCollection valueColl =mm.Values;
Console.WriteLine();
foreach (string s in valueColl)
{
    Console.WriteLine("Value = {0}", s);
}
Dictionary<string, string>.KeyCollection keyColl =mm.Keys;
Console.WriteLine();
foreach (string s in keyColl)
{
    Console.WriteLine("Key = {0}", s);
}
Console.WriteLine("\nRemove(\"ggg\")");
mm.Remove("ggg");
if (!mm.ContainsKey("ggg"))
{
```

```
                    Console.WriteLine("没有找到键 \"ggg\" 。");
                }
                Console.ReadKey();
            }
        }
    }
```

上述实例代码的设计流程如下。

1）定义类型键值对象 mm。

2）通过 Add()分别向 mm 内添加键值 aaa、bbb、ccc 和 ddd。

3）如果再次使用 Add()方法向 mm 内添加键 aaa，则输出对应提示。

4）输出显示对象 mm 内的对应键值。

5）使用索引器来赋值 ddd，如果键已经存在，那么就会修改对应的值。

6）使用索引器来赋值，如果键不存在则添加这个元素值。

7）使用索引器来获取值时，如果此键不存在则会引发异常，并输出对应提示。

8）通过 TryGetValue ()方法获取键值，如果不存在则输出对应的异常提示。

9）通过 ContainsKey()方法进行插入判断，如已经存在则输出对应提示。

10）通过 Remove()删除 mm 内指定的键值。

11）输出并显示各种操作后 mm 内的键值。

执行后将根据编写的操作方法对 mm 对象的键值进行操作处理，并输出显示处理后的对应数据，如图 13−2 所示。

```
有一个元素已经使用了键 = "aaa"。
对应键 "ddd" 的值为 d.exe。
对应键 "rtf" 的值为 dd.exe。
没有找到对应键 "eee" 的值。
没有找到对应键 "eee" 的值。
已经添加了对应键 "fff" 的值: hypertrm.exe

Key = aaa, Value = a.exe
Key = bbb, Value = b.exe
Key = ccc, Value = b.exe
Key = ddd, Value = dd.exe
Key = ggg, Value = dd.exe
Key = fff, Value = hypertrm.exe
```

图 13−2　实例执行结果

在上述实例代码中，为了避免键值不存在所引发的异常，使用了异常处理机制对程序进行优化。



多学一招

泛型的最大优点是类型安全转换，值类型存储在堆栈上，引用类型存储在堆上。C#类是引用类型，结构是值类型。.NET 很容易把值类型转换为引用类型，所以可以在需要对象(对象是引用类型)的任意地方使用值类型。例如，int 可以赋予一个对象。从值类型转换为引用

类型称为装箱。如果方法需要把一个对象作为参数，而且传送了一个值类型，装箱操作就会自动进行。另一方面，装箱的值类型可以通过拆箱操作转换为值类型。在拆箱时，需要使用类型转换运算符。

装箱和拆箱操作很容易使用，但性能损失比较大，迭代许多项时尤其如此。这时，就可以考虑使用泛型来高效、安全地实现功能。另外，泛型允许更好地重用二进制代码。泛型类可以定义一次，用许多不同的类型实例化。不需要像 C++模板那样访问源代码。例如，System.Collections.Generic 命名空间中的 List<T>类用一个 int、一个字符串和一个 MyClass 类型实例化:

```
List<int> list = new List<int>();
list.Add(44);
List<string> stringList = new List<string>();
stringList.Add("mystring");
List<MyClass> myclassList = new List<MyClass>();
myClassList.Add(new MyClass());
```

13.4.3　自定义泛型集合类

在系统项目应用中，用户可以自行定义需要的泛型接口、泛型类、泛型方法、泛型事件和泛型委托，以便实现系统的特殊功能需求。在 C#中，允许程序员自定义泛型集合类，但前提是定义的集合类必须实现 System.Collections.Generic 命名空间中的 IEnumerable<T>和 IEnumerator<T>的接口。

因为 IEnumerable<T>和 IEnumerator<T>分别继承了 IEnumerable 和 IEnumerator 接口，所以在实现 IEnumerable<T>接口类的同时，也必须实现 IEnumerable 接口；而在实现 IEnumerator<T>接口的同时，必须同时实现 IEnumerator 接口。要实现某类的 IEnumerable<T>接口，必须使用如下格式的代码：

```
System.Collections.Generic.IEnumerator<T>
  System.Collections.Generic.IEnumerable<T>.GetEnumerator();
System.Collections.Generic.IEnumerator
  System.Collections.Generic.IEnumerable.GetEnumerator();
```

而如果想实现某类的 IEnumerator<T>接口，必须使用如下格式的代码：

```
bool System.Collections.IEnumerator.MoveNext();
void System.Collections.IEnumerator.Reset();
object System.Collections.IEnumerator.Current;
T System.Collections.IEnumerator.Generic.Current;
```

非泛型成员可以通过调用同名泛型成员来实现。

实例 60：自定义泛型集合类实现数据处理

下面将通过一个简单实例来说明使用自定义泛型集合类实现数据处理的过程。本实例保存在“光盘：\daima/13”文件夹内，项目名为 dingyi。本实例的功能是，向对象列表内添加

数据，并输出显示处理后的列表数据。实例文件的主要代码如下所示：

```
namespace zidingyi {
    public class mm<T> : IEnumerable<T>, IEnumerator<T>
    {
        private string aa = "索引无效";
        private int index = -1;
        protected List<T> qiuduis;
        public mm()
        {
            qiuduis = new List<T>();
        }
        public void Add(T qiudui)
        {
            qiuduis.Add(qiudui);
        }
        public void Remove(T qiudui)
        {
            qiuduis.Remove(qiudui);
        }
        public System.Collections.Generic.IEnumerator<T> GetEnumerator()
        {
            Reset();
            return this;
        }
        // IEnumerable<T> 继承了 IEnumerable，因此本类必须同时实现泛型和非泛型
版本的 GetEnumerator
        // 一般来说，非泛型方法可以简单地调用泛型 GetEnumerator 方法
        System.Collections.IEnumerator System.Collections.IEnumerable.
GetEnumerator()
        {
            return GetEnumerator();
        }
        public bool MoveNext()
        {
            return (++index < qiuduis.Count);
        }
        public void Reset()
        {
            index = -1;
        }
        public T Current
        {
            get
            {
                if (0 <= index && index < qiuduis.Count)
                    return qiuduis[index];            // 返回当前元素
```

```
                else                                           // 索引无效
                    throw new IndexOutOfRangeException(aa);  // 抛出异常
            }
        }
        // IEnumerator<T> 继承了 IEnumerator，因此本类必须同时实现泛型和非泛型
    版本的 Current
        // 一般来说，非泛型方法可以简单地调用泛型的 Current 属性
        object System.Collections.IEnumerator.Current
        {
            get
            {
                return Current;
            }
        }
        public void Dispose()
        {
        }
    }
    public class Test {
        public static void Main()
        {
            mm<string> club = new mm<string>();
            club.Add("AA");
            club.Add("BB");
            club.Add("CC");
            club.Add("DD");
            club.Add("EE");
            Console.WriteLine("我喜欢的有：\n");
            foreach (string qiudui in club)
            {
                Console.WriteLine(qiudui + "\n");
            }
            Console.ReadKey();
        }
    }
}
```

上述实例代码的设计流程如下。

1）通过 using 调用 System.Collections.Generic 类，定义命名空间 zidingyi。

2）自定义泛型类 mm，设置同时实现和 IEnumerabl、IEnumerator 的接口。

3）设置变量 aa 和 index 的初始值。

4）如果当前索引无效则输出变量 aa 的错误提示。

5）定义类 mm 的构造函数，并设置其 ArrayList 类型成员 qiuduis，用于存储数据。

6）分别实现方法 Rset()、MoveNext()和 GetEnumerator()的处理。

7）定义自定义类 mm 的成员 club，用于存储数据。

8）通过方法 Add()向 club 内添加五个数据。

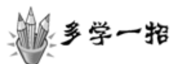
9）使用 foreach 将当前的数据输出显示。

上述实例代码执行后，将输出显示添加处理后的数据，具体如图 13−3 所示。

在上述代码中，因为 IEnumerable<T>继承了 IEnumerable，所以类 qiudui 必须实现泛型和非泛型版本的 Get IEnumerator。并且在实现非泛型版本的 Get IEnumerator 和 Current 时，使用显式接口成员。

```
我喜欢的有：
AA
BB
CC
DD
EE
```

图 13−3 实例执行结果



因为泛型类的定义会放在程序集中，所以用某个类型实例化泛型类不会在 IL 代码中复制这些类。但是，在 JIT 编译器把泛型类编译为内部代码时，会给每个值类型创建一个新类。引用类型共享同一个内部类的所有实现代码。这是因为引用类型在实例化的泛型类中只需要 4 字节的内存单元(32 位系统)，就可以引用一个引用类型。值类型包含在实例化的泛型类的内存中。而每个值类型对内存的要求都不同，所以要为每个值类型实例化一个新类。

13.5 C#泛型迭代器

和普通集合类一样，在 C#中也可以使用泛型迭代器来实现泛型集合。泛型迭代器和非泛型迭代器的用法类似，唯一的区别是泛型迭代器的返回类型必须是 IEnumerable<T>和 IEnumerator<T>。在本节的内容中，将通过一个具体实例的实现过程，向读者讲解 C#泛型迭代器的使用过程。

实例 61：使用泛型迭代器处理列表

本实例保存在“光盘：\daima\13”文件夹内，项目名为 diedaiqi。本实例的功能是，通过泛型迭代器对定义的集合对象进行处理，并输出显示处理后的列表数据。实例文件的主要代码如下：

```
namespace diedaiqi
{
   public class mm<T>
   {
      protected List<T> qiuduis;
      public mm()
      {
         qiuduis = new List<T>();
      }
      public void Add(T qiudui)
      {
         qiuduis.Add(qiudui);
      }
      public void Remove(T qiudui)
```

```
            {
                qiuduis.Remove(qiudui);
            }
            public IEnumerator<T> GetEnumerator()
            {
                foreach (T qiudui in qiuduis)
                {
                    yield return qiudui;
                }
            }
        }
        public class Test
        {
            public static void Main()
            {
                mm<string> club = new mm<string>();
                club.Add("AA");
                club.Add("BB");
                club.Add("CC");
                club.Add("DD");
                club.Add("EE");
                Console.WriteLine("我喜欢的有：\n");
                foreach (string qiudui in club)
                {
                    Console.WriteLine(qiudui + "\n");
                }
                Console.ReadKey();
            }
        }
    }
```

上述实例代码的设计流程如下。

1）通过 using 调用 System.Collections.Generic 类，定义命名空间 diedaiqi。

2）自定义泛型类 mm，设置类型为 T。

3）定义类 mm 的构造函数，并设置其 ArrayList 类型成员 qiuduis，用于存储数据。

4）分别实现方法 Add()、Rmove()和 GetEnumerator()的操作处理。

5）通过 yield 迭代，返回对象集合列表内的数据。

6）定义自定义类 mm 的成员 club，用于存储数据。

7）通过方法 Add()向 club 内添加五个数据。

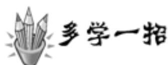
8）使用 foreach 将当前的数据输出显示。

执行后将输出显示操作处理后的数据，如图 13-4 所示。

由此可以看出，通过泛型迭代器可以同样操作对象的数据，并且减少了代码编写量。

```
我喜欢的有：
AA
BB
CC
DD
EE
```

图 13-4　实例执行结果



在 C#1.0 中，若想使类型成为可枚举的类型(即如同使用数组一样使用 foreach 枚举数组中的成员)，那么这个类中必须实现 IEnumerable 接口的 GetEnumerator()方法返回 IEnumerator:

```
        public IEnumerator GetEnumerator()
        {..}
```

一般来说实现 GetEnumerator 是不容易的，从 C#2.0 开始引入了的新的语法扩充——迭代器，它简单方便地实现 GetEnumerator()方法。

另外，从 C#2.0 开始引入了一个“不全代码”的概念，即可以将一个类的定义放在别的文件中。这样给大项目、大文件的维护带来了方便。不全代码的声明格式如下:

```
        [modifiers] partial type
        modifier:  [public/private/protected/internal]  abstract  new  override
static virtual extern
        type:类/结构/接口
```

13.6 疑难问题解析

本章详细介绍了 C#泛型的基本知识。本节将对本章中比较难以理解的问题进行讲解。

读者疑问：使用 C#泛型有什么好处？

解答：使用 C#泛型后的优点主要如下:

- 泛型将类型参数的概念引入到了.NET 框架，类型参数使泛型类和方法将确定一个或多个类型的时间推迟到客户端代码声明并实例化类执行的时候。并且不会引入强制转化成本，也不会存在错误风险。
- 泛型类和泛型方法同时具有可重用性，类型安全有效率。

读者疑问：C#泛型可以完成单例模式的重用吗？

解答：使用泛型可以完成单例模式的重用。使用泛型后，使得实现“单例提供者”成为可能。这是一个可用来创建单例类实例却不需要为每个特定的类重写单例模式代码的可重用的类。这样分离出单例结构的代码，将有利于保持按单例模式使用类或不按单例模式使用类的灵活性。

```
        /// <summary>
           /// 单例提供者
           /// </summary>
           /// <typeparam name="T">需要的类</typeparam>
           public class SingletonProvider<T> where T : new()
           {
               SingletonProvider() { }
```

```
            public static T Instance
            {
                get { return SingletonCreator.instance; }
            }
            class SingletonCreator
            {
                static SingletonCreator() { }
                internal static readonly T instance = new T();
            }
        }
```

在定义泛型类时，可以对客户端代码能够在实例化类时用于类型参数的类型种类施加限制。如果客户端代码尝试使用某个约束所不允许的类型来实例化类，则会产生编译错误。这些限制称为约束。约束是使用 where 上下文关键字指定的。下面列出五种类型的约束:

- 结构，类型参数必须是值类型，可以指定除 Nullable 以外的任何值类型。
- 类，类型参数必须是引用类型，包括任何类、接口、委托或数组类型。
- new()，类型参数必须具有无参数的公共构造函数。当与其他约束一起使用时，new() 约束必须最后指定。
- <基类名>，类型参数必须是指定的基类或派生自指定的基类。
- <接口名称>，类型参数必须是指定的接口或实现指定的接口。可以指定多个接口约束。约束接口也可以是泛型的。
- U，为 T 提供的类型参数必须是为 U 提供的参数或派生自为 U 提供的参数。这称为裸类型约束。



职场点拨——认清你的职责

职场中的你，可能会遇见这样或那样的任务。很多读者可能会想，作为一个新人，多做一点能得到领导的赏识。在领导面前表现自己并没有错，但是并不意味着做得越多越好。俗话说物极必反，在认真工作的前提下一定要认清自己的职责。具体来说，应遵循下面的四条原则:

1）公司招聘员工，是给员工一个工作岗位的，这个岗位对应了一个工作，你的任务就是完成自己本职工作。在完成应该做的本职工作后，如果有充足的时间，可以去干其他的工作。

2）即使去干其他工作，也要有主次之分，即只做次要的、不影响大局的。这样既帮助了同事，也不会有喧宾夺主的感觉，能够加深同事间的感情。

3）确保能很好地完成其他工作，并且保证能够出色地完成。不然会白忙活一场，不但浪费了同事的时间，而且也会在领导面前形成不好的印象。

4）如果同事再三要求需要帮忙，一定要确认能否胜任。如果不能，要提前向他声明，并且最好有第三方作证。要是提前和上级打好招呼就最好了。

第 14 章　文 件 操 作

在计算机系统中，文件操作又被称为 I/O 操作，能够实现输入和输出操作处理。文件操作功能是编程语言所要面对的主要课题之一，C#能够根据系统的特殊需求，对文件进行新建、删除、读写和更新等操作。在.NET 框架中，提供了功能强大的文件操作接口。在.NET 框架类库的 System.IO 的命名空间中，提供了多个用于文件操作的类型。C#通过上述类型，可以方便地实现对文件的各种操作处理。通过本章能学到如下知识。

- 文件管理。
- C#流文件管理。
- XML 文件处理。
- 职场点拨——程序员的几个坏习惯。

2008 年 X 月 XX 日，晴

今天大学老师 A 找到我，让我做一个学校图书馆项目。我心想不错，不单可以练练技术，而且还能赚一笔外快。我决定马上开工，经过 3 个日夜的埋头苦干，终于完成了。老师看后说："小菜同学，你遗漏了很多功能啊，是不是开工前没做规划啊？"



一问一答

小菜：郁闷死我了，我竟然遗漏了一个功能模块！

Wisdom：呵呵，在做一个项目之前，最好统筹规划，做好项目分析，画一个结构图，这样就不会遗漏了，并且还能对进度有一个大体的把握！

小菜：每个项目都得事先规划吗？

Wisdom：当然了，你现在在公司的身份是基层程序员，当有新项目时，项目经理会规划整个项目，并统筹安排人力分工，所以规划的事情不用你操心。但是你现在做的是一个私人项目，要你一个人完成，就需要你来做前期规划工作了。

小菜：嗯，以后我会注意的！

Wisdom：你必须得注意！表面看是小问题，但是日积月累，就会养成坏的习惯。很多专家箴言，程序员要养成好的习惯，这才是成功的前提，才能有更好的发展前景。在本章的最后，我介绍了几个程序员容易养成的坏习惯，建议你去看看！

小菜：嗯，言归正传，本章所讲的文件操作很重要吗？

Wisdom：精通文件操作后，你就将正式迈入程序员的行列。不管使用什么语言，都难免要接触到文件系统，要经常和文件打交道，C#当然也不例外。实现文件操作不但需要对基本语法精通掌握，还需要对函数功能有所造诣，因为文件的处理就是通过对应的函数来实现的。

14.1　文件管理

在.NET 框架类库 System.IO 的命名空间内，提供了专用的类来实现对文件系统的操作管理，包括常见的复制、删除、文件移动等操作。在 System.IO 命名空间中，和文件管理类相关的主要信息如下。

- System.MarshallByRefObject：访问远程处理的应用程序，可以在不同分布的程序之间调用数据。
- System.IO.FileSystemInfo：类 FileInfo 和 DirectoryInfo 的基类，封装了文件和目录操作的全部方法。
- System.IO.File：提供了创建、复制、删除移动等文件处理的静态方法，并协同创建了对象 FileStream。
- System.IO.FileInfo：提供了创建、复制、删除移动等文件处理的静态方法，并协同创建了对象 FileStream。
- System.IO.Directory：提供了创建、复制、删除移动等文件处理的实例方法，并协同创建了对象 FileStream。
- System.IO.DirectoryInfo：提供了创建、移动和枚举目录和子目录的实例方法。
- System.IO.Path：对包含文件或目录路径的 String 实例进行跨平台方式操作。
- System.Environment：设置当前运行环境平台的信息。

上述各个文件操作类型的对应关系如图 14-1 所示。

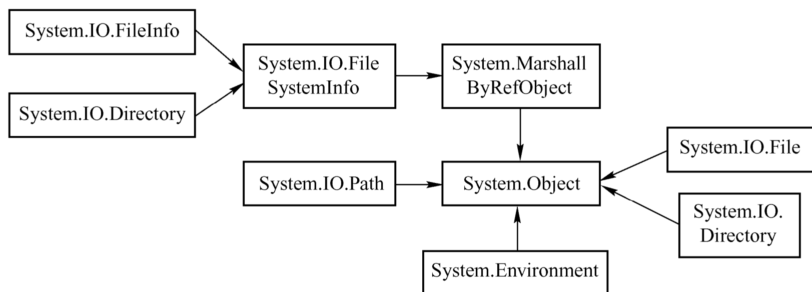


图 14-1　System.IO 命名空间文件操作类关系图

因为上述各个文件操作类型都是静态类或封闭类，所以不能被继承。

14.1.1　使用 File 和 FileInfo

IO 内的 File 和 FileInfo 类的功能类似，能够实现对文件的复制、移动、重命名、创建和打开等基本操作，并能获取和设置文件属性及文件创建、访问和写入操作的 DateTime 信息。类 File 中主要方法的具体说明如表 14-1 所示。

表 14-1　类 File 方法信息

方　　法	说　　明
AppendAllText（）	将指定字符串添加到文件中，如果文件不存在则创建该文件

（续）

方　　法	说　　明
AppendText（）	创建一个 Stream，将 WriteUTF-8 编码文件进行追加
Copy（）	将现有文件复制到新文件
Create（）	创建新文件
CreateText（）	创建或打开一个 UTF-8 编码的文件
Decrypt（）	解密用 Encrypt 加密的文件
Delete（）	删除指定文件
Encrypt（）	加密某文件
Exists（）	验证文件是否存在
GetAccessControl（）	获取一个 FileSecurity 对象
GetAttributes（）	获取文件的属性
GetCreationTime（）	获取文件或目录的创建时间
GetCreationTimeUtc（）	获取文件或目录的创建 UTC 格式时间
GetLastAccessTime（）	获取文件或目录的最后被访问时间
GetLastAccessTimeUtc（）	获取文件或目录的最后被访问的 UTC 格式时间
GetLastWriteTime（）	获取文件或目录的上次被写入的时间
GetLastWriteTimeUtc（）	获取文件或目录的上次被写入的 UTC 格式时间
Move（）	移动某文件
Open（）	打开某路径上的 FileStream
OpenRead（）	打开文件并读取
OpenText（）	打开 UTF-8 编码文件并读取
OpenWrite（）	打开文件并写入
ReadAllBytes（）	打开文件并将内容读入一个字符串，然后将文件关闭
ReadAllLines（）	打开文件并将所有行读入一个字符串数组，然后将文件关闭
ReadAllText（）	打开文件并将所有行读入一个字符串，然后将文件关闭
Replace（）	替换某文件内容
SetAccessControl（）	对指定文件应用由 FileSecurity 对象描述的访问控制列表项
SetAttributes（）	设置某文件指定的 FileAttributes
SetCreationTime（）	设置创建文件的时间
SetCreationTimeUtc（）	设置创建文件的 UTC 格式时间
SetLastAccessTime（）	设置上次访问文件的时间
SetLastAccessTimeUtc（）	设置上次访问文件的 UTC 格式时间
SetLastWriteTime（）	设置上次写入文件的时间
SetLastWriteTimeUtc（）	设置上次写入文件的 UTC 格式时间
WriteAllBytes（）	创建新文件，并写入指定的字节数组，然后关闭文件
WriteAllLines（）	创建新文件，并写入指定的字符串，然后关闭文件
WriteAllText（）	创建新文件，并写入内容，然后关闭文件

类 FileInfo 的主要属性信息如表 14-2 所示。

表 14-2　类 FileInfo 属性信息

属　　性	说　　明
Attributes	获取设置当前 FileSystemInfo 的 FileAttributes
CreationTime	获取设置当前 FileSystemInfo 的创建时间
CreationTimeUtc	获取设置当前 FileSystemInfo 的 UTC 格式时间
Directory	解密用 Encrypt 加密的文件
DirectoryName	获取父目录实例
Exists	验证文件是否存在
Extension	获取文件扩展名字符串
FullName	获取文件或目录的完整名或目录
IsReadOnly	设置文件为只读
LastAccessTime	获取文件或目录的上次被访问的时间
LastAccessTimeUtc	获取文件或目录的上次被访问的 UTC 格式时间
LastWriteTime	获取文件或目录的上次被写入的时间
LastWriteTimeUtc	获取文件或目录的上次被写入的 UTC 格式时间
Length	获取文件的大小
Name	获取文件名

类 FileInfo 的主要方法信息如表 14-3 所示。

表 14-3　类 FileInfo 方法信息

方　　法	说　　明
AppendText（）	创建一个 StreamWriter
CopyTo（）	将现有文件复制到新文件
Create（）	创建新文件
CreateObjRef（）	创建远程代理对象
CreateText（）	创建、写入新文件的 StreamWriter
Decrypt（）	解密用 Encrypt 加密的文件
Delete（）	删除指定文件
Encrypt（）	加密某文件
GetAccessControl（）	获取 FileSecurity 对象
MoveTo（）	移动某文件
Open（）	打开某文件
OpenRead（）	创建只读 FileStream
OpenText（）	打开 UTF-8 编码文件并读取
OpenWrite（）	打开只写 FileStream
Replace（）	替换某文件内容
SetAccessControl（）	对指定文件应用由 FileSecurity 对象描述的访问控制列表项

在读写文件时，通过指定字符的编码方式，来明确说明被读写文件是基于哪种字符编码进行的。编码能够将一组 Unicode 字符转换为一个字节顺序，而解码则是上述过程的反

向操作。

通过 Unicode 标准，可以为所有支持脚本中的每个字符分配一个码位，Unicode 转换格式 UTF 是一种码位的编码方式。Unicode 标准使用如下三种 UTF。

1）UTF-8：将每个码位表示为一个由 1~4 个字节组成的序列。

2）UTF-16：将每个码位表示为一个由 1~2 个 16 位整数组成的序列。

3）UTF-32：将每个码位表示为一个 32 位整数序列。

因为所有的 File 类都是静态的，所以如果想要实现对某文件的操作，使用 File 的效率会比使用 FileInfo 实例方法的要高。看下面的一段代码：

```
using System.IO;
namespace file111111{
    public class FileTest
    {
        public static void Main()
        {
            string m_path = @"e:\123.txt";      //设置所要创建文件的绝对路径
            File.Create(m_path);                //以路径为参数创建文件
            Console.ReadLine();
        }
    }
}
```

运行上述代码后，会在 E 盘创建一个名为“123.txt”的文件。

使用 FileInfo 实例时需要传入当前操作的文件路径。如果想多次重用某个对象，则需要使用 FileInfo 的实例方法，而不是使用 File 类的静态方法。看下面的一段代码：

```
namespace FileInfolizi{
    class Program
    {
        public static void Main()
        {
            string mm = @"c:\123.txt";          //设置所要检测文件的绝对路径
            FileInfo nn = new FileInfo(mm);//以路径为参数构造 FileInfo 对象
            if (nn.Exists)                      //检测文件是否存在
            {
                Console.WriteLine("已经存在!");
            }
            else
            {
                Console.WriteLine("不存在!");
            }
            Console.ReadLine();
        }
    }
}
```

在上述代码中，验证在 C 盘中是否存在指定的文件“123.txt”，如果不存在则输出“不存在!”；如果已经存在，则输出“已经存在!”。

14.1.2　使用 Directory 和 DirectoryInfo

Directory 类和 DirectoryInfo 类的功能类似，主要功能如下。

1）实现基本的文件复制、移动、重命名、创建和删除等基本操作。

2）列举目录中的文件和子目录。

3）获取和设置目录信息，实现目录创建、访问和写入等操作相关的时间信息。

类 Directory 的主要方法信息如表 14−4 所示。

表 14−4　类 Directory 方法信息

方　　法	说　　明
CreateDirectory（）	创建某路径内的所有目录
Delete（）	删除指定目录
Exists（）	验证目录是否存在
GetAccessControl（）	获取某目录的 Windows 访问控制列表
GetCreationTime（）	获取目录的创建时间
GetCreationTimeUtc（）	获取目录的创建 UTC 格式时间
GetCurrentDirectory（）	获取当前运行程序的目录
GetDirectories（）	获取某目录内子目录的名称
GetDirectoryRoot（）	获取指定路径卷信息和根信息
GetFiles（）	获取返回目录的文件名
GetFileSystemEntries（）	获取指定目录的所有文件和子目录信息
GetLastAccessTime（）	获取文件或目录的最后被访问时间
GetLastAccessTimeUtc（）	获取文件或目录的最后被访问的 UTC 格式时间
GetLastWriteTime（）	获取文件或目录的上次被写入的时间
GetLastWriteTimeUtc（）	获取文件或目录的上次被写入的 UTC 格式时间
GetLogicalDrives（）	检索机器上主驱动磁盘的名称
GetParent（）	检索指定路径的父目录
Move（）	移动某文件或目录
SetAccessControl（）	对指定目录应用由 FileSecurity 对象描述的访问控制列表项
SetCreationTime（）	设置创建文件或目录的时间
SetCreationTimeUtc（）	设置创建文件或目录的 UTC 格式时间
SetCurrentDirectory（）	为指定文件或目录设置日期和时间
SetLastAccessTime（）	设置上次访问文件或目录的时间
SetLastAccessTimeUtc（）	设置上次访问文件或目录的 UTC 格式时间
SetLastWriteTime（）	设置上次写入文件的时间
SetLastWriteTimeUtc（）	设置上次写入文件或目录的 UTC 格式时间

类 DirectoryInfo 的主要属性信息如表 14−5 所示。

表 14-5　类 DirectoryInfo 属性信息

属　　性	说　　明
Attributes	获取设置当前 FileSystemInfo 的 FileAttributes
CreationTime	获取设置当前 FileSystemInfo 的创建时间
CreationTimeUtc	获取设置当前 FileSystemInfo 的 UTC 格式时间
Exists	验证目录是否存在
Extension	获取文件扩展名字符串
FullName	获取文件或目录的完整目录
LastAccessTime	获取文件或目录的上次被访问的时间
LastAccessTimeUtc	获取文件或目录的上次被访问的 UTC 格式时间
LastWriteTime	获取文件或目录的上次被写入的时间
LastWriteTimeUtc	获取文件或目录的上次被写入的 UTC 格式时间
Name	获取此 DirectoryInfo 实例的名称
Parent	获取指定目录的父目录
Root	获取路径的根

类 DirectoryInfo 的主要方法信息如表 14-6 所示。

表 14-6　类 DirectoryInfo 方法信息

方　　法	说　　明
Create（）	创建目录
CreateObjRef（）	创建远程代理对象
CreateSubdirectory（）	在指定路径下创建一个或多个子目录
Delete（）	删除 DirectoryInfo 实例和内容
GetAccessControl（）	获取当前目录的访问控制列表项
GetDirectories（）	获取某目录内子目录的名称
GetFiles（）	获取返回目录的子目录
GetFileSystemInfos（）	检索指定目录的所有文件和子目录的强类型 FileSystem 对象数组
MoveTo（）	移动 DirectoryInfo 实例
SetAccessControl（）	对指定目录应用由 DirectorySecurity 对象描述的访问控制列表项

使用类 Directory 的方法和 File 的类似，因为所有的 Directory 类方法都是静态的，所以当只执行一个操作时，使用 Directory 方法的效率会高于使用 DirectoryInfo 实例方法的效率。看下面一段代码：

```
using System.IO;
namespace directory123
{
    class Program
    {
        public static void Main()
        {
```

```
                string m_path = @"e:\123.txt";//设置所要创建文件夹的绝对路径
                Directory.CreateDirectory(m_path); //以路径为参数创建文件夹
                Console.ReadLine();
            }
        }
    }
```

上述代码执行后，将会在 E 盘内创建一个名为“123.txt”的文件。

使用 DirectoryInfo 类的方法和使用 FileInfo 类的方法类似。在使用 DirectoryInfo 时，必须先构造 DirectoryInfo 实例，并传入当前操作目录的路径。如果在项目中需要多次重用某个对象，最好使用 DirectoryInfo 实例方法，而不使用 Directory。看下面的一段代码：

```
    using System.IO;
    namespace DirectoryInfo123{
        class Program
        {
            static void Main(string[] args)
            {
                string m_path = @"e:\123";     //设置所要检测文件夹的绝对路径
                //以路径为参数构造 DirectoryInfo 对象
                DirectoryInfo m_DirInfo = new DirectoryInfo(m_path);
                if (m_DirInfo.Exists)          //检测文件夹是否存在
                {
                    Console.WriteLine("已经存在!");
                }
                else
                {
                    Console.WriteLine("还没有存在!");
                }
                Console.ReadLine();
            }
        }
    }
```

执行后将会在 E 盘内检查是否存在名为“123”的文件夹，并输出对应的提示，如图 14-2 所示。

已经存在!

图 14-2　实例执行结果

如果在某个目录内存在多个文件或子目录，使用 Directory 和 DirectoryInfo 类将不能删除这个目录。下面通过一个具体实例来具体讲解解决上述限制的办法。

实例 62：使用 Directory 和 DirectoryInfo 的方法和属性

本实例保存在“光盘:\daima\14”文件夹内，项目名为 DirectoryDirectoryInfo。功能是使用 Directory 和 DirectoryInfo 的方法和属性，删除“e:\123”目录。其中“e:\123”目录下有

三个子目录，如图 14-3 所示。

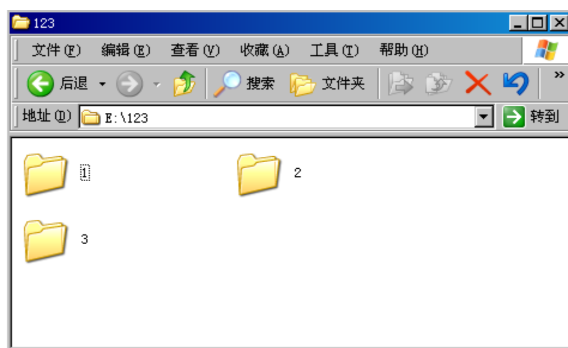


图 14-3 “123”目录下的子目录

实例文件 Program.cs 的主要代码如下：

```
using System.IO;
namespace DirectoryDirectoryInfo{
    class Program
    {
        public static void Main(string[] args)
        {
            string dir = "e:\\123";
            if (dir.Length == 0)
            {
                Console.WriteLine("请指定要删除的目录名。");
                Console.ReadKey();
                return;
            }
            else
            {
                dir = dir;
            }
            DirectoryInfo di = null;
            try
            {
                di = new DirectoryInfo(dir);
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
            DeleteFiles(di);
        }
        public static void DeleteFiles(DirectoryInfo di)
```

```
            {
                try
                {
                    FileInfo[] files = di.GetFiles();
                    foreach (FileInfo fin files)
                    {
                        f.Delete();
                    }
                    DirectoryInfo[] dirs = di.GetDirectories();
                    foreach (DirectoryInfo din dirs)
                    {
                        if (d.Exists)
                        {
                            DeleteFiles(d);
                        }
                    }
                    di.Delete();
                }
                catch (Exception ex)
                {
                    Console.WriteLine(ex.Message);
                }
            }
        }
    }
```

上述实例代码的设计流程如下。

1）定义变量 dir，设置删除的目录是“e:\123”。

2）如果 dir 为空则输出提示。

3）通过 FileInfo 实例 files 获取目录中的文件。

4）通过 foreach 语句删除目录内的所有文件。

5）通过 DirectoryInfo 实例 dirs，获取目录内的所有子目录。

6）通过 foreach 语句删除目录内的所有子目录。

7）当子目录和子文件删除完毕后，则将“e:\123”文件夹删除。

执行后首先默认显示控制台界面，按〈Enter〉键后将删除对应的子目录，删除完毕后退出控制台界面。

14.1.3　使用 Path

通过使用类 Path，可以操作文件和其包含目录信息字符串的信息，例如常见的扩展名信息验证和检测等。因为类 Path 内的所有成员都是静态的，所以不用路径实例即可被调用，并且类 Path 可以跨平台操作。

类 Path 中的主要字段信息如表 14-7 所示。

表 14-7　类 Path 字段信息

字　　段	说　　明
AltDirectorySeparatorChar	提供平台特定的替换字符，该替换字符用于在反映层文件系统组织的路径字符串中分割目录的级别
DirectorySeparatorChar	提供平台特定字符，该字符用于反映分层文件系统组织的路径字符串中分割目录级别
InvalidPathChars	提供平台特定的字符串数组，这些字符不能在传递 Path 类的成员路径字符串参数中指定
PathSeparator	用于环境变量中分割路径字符串的平台特定分隔符
VolumeSeparatorChar	提供平台特定的卷分隔符

类 Path 中的主要方法信息如表 14-8 所示。

表 14-8　类 Path 方法信息

方　　法	说　　明
ChangeExtension（）	更改路径字符串的扩展名
Combine（）	合并两个路径字符串
GetDirectoriyName（）	返回指定路径字符串的目录信息
GetExtension（）	返回指定路径字符串的扩展名
GetFileName（）	返回指定路径字符串的文件名和扩展名
GetFileNameWithoutExtension（）	返回不具有扩展名的指定路径字符串的文件名
GetFullPath（）	返回指定路径字符串的绝对路径
GetInvalidFileNameChars（）	获取包含不允许在文件名中使用字符的数组
GetInvalidPathChars（）	获取包含不允许在路径名中使用字符的数组
GetPathRoot（）	获取指定路径的根目录信息
GetRandomFileName（）	返回随机文件夹或文件名
GetTempFileName（）	在磁盘内创建唯一名的 0 字节的临时文件
GetTempPath（）	返回当前系统的临时文件夹路径
HasExtension（）	确定路径是否包含文件扩展名
IsPathRooted（）	获取指示某路径字符串是否包含绝对路径信息还是相对路径信息的值

类 Path 中的大多数成员不能和文件系统进行交互，不能验证某路径字符串上的文件是否存在。所以如果修改类 Path 的成员后，不会影响系统文件的名称。



路径是提供文件或目录位置的字符串，它不必执行磁盘上的某个位置，例如路径可以映射到内存中以获取设备上的位置。路径的准确格式和使用平台有关，不同平台的路径标准不同。并且文件扩展名的格式也和平台有关。

实例 63：使用 Path 类

下面将通过一个具体的实例来说明使用 Path 类的具体流程。本实例保存在“光盘：\daima\14”文件夹内，项目名为 Pathshiyong。功能是使用 Path 类分析指定路径，实现文件的主要代码如下：

```
namespace Pathshiyong {
```

```
    class Program
    {
        public static void Main(string[] args)
        {
            string path = "e:\\c#\\daima\\14\\Pathchuli\\Program.cs";
            if (path.Length == 0)
            {
                Console.WriteLine("请指定要分析路径名。");
                Console.ReadKey();
                return;
            }

            string cipan, mulu, file, ext;
            ParsePath(path, out cipan, out mulu, out file, out ext);
            Console.WriteLine("磁盘：" + cipan);
            Console.WriteLine("\n 目录名：" + mulu);
            Console.WriteLine("\n 文件名：" + file);
            Console.WriteLine("\n 扩展名：" + ext);
            Console.WriteLine("\n\n 重新组合的路径："+MakePath(cipan,mulu,file));
            Console.ReadKey();
        }
        public static void ParsePath(string path, out string cipan, out
string mulu, out string file, out string ext)
        {
            cipan = "";
            mulu = "";
            file = "";
            ext = "";
            try
            {
                cipan = Path.GetPathRoot(path);
                mulu = Path.GetDirectoryName(path);
                file = Path.GetFileName(path);
                ext = Path.GetExtension(path);
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
        }
        public static string MakePath(string cipan, string mulu, string file)
        {
            string path = "";
            try
            {
                string path1 = Path.Combine(cipan, mulu);
```

```
                path = Path.Combine(path1, file);
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
            return path;
        }
    }
}
```

上述实例代码的设计流程如下。

1）定义变量 path，设置被分析文件的路径是“e:\c#\daima\14\Pathchuli\Program.cs。

2）如果 path 为空，则输出提示。

3）分别定义 string 类型变量 cipan、mulu、file 和 ext，分别用于获取文件所述的磁盘名、所属目录名、文件名和扩展名。

4）通过 WriteLine()输出显示分析后的结果。

执行后将输出对 path 指定文件分析后的结果。如果 path 指定文件为空，则输出对应的不存在提示，如图 14-4 所示。

```
磁盘：e:\
目录名：e:\c#\daima\14\Pathchuli
文件名：Program.cs
扩展名：.cs

重新组合的路径：e:\c#\daima\14\Pathchuli\Program.cs
```

图 14-4　输出分析结果


多学一招

在使用 Path 类时要注意如下两点。

1）类 Path 的大多数成员不能和文件系统进行交互，不能验证某路径字符串上的文件的存在。所以如果对类 Path 的成员进行修改，不会影响系统文件的名称。

2）路径是提供文件或目录位置的字符串，它不必执行磁盘上的某个位置。例如路径可以映射到内存中设备上的位置，路径的准确格式和使用平台有关，不同平台的路径标准不同。并且文件扩展名的格式也和平台有关。

14.1.4　使用 Environment

类 Environment 的功能是获取当前系统平台和运行环境的基本信息，它在 System 中定义。因为类 Environment 是一个静态类，所以不能够被继承。通过 System.Environment 类，能够检索和操作环境相关的大量信息，例如命令参数、环境变量设置和系统上次启动的时间等信息。类 Environment 的主要属性如表 14-9 所示。

表 14-9 类 Environment 属性信息

属　　性	说　　明
CommandLine	获取当前进程命令行
CurrentDirectory	获取设置当前目录的完全限定名
ExitCode	获取设置进程的退出代码
HasShutddownStarted	获取是否只是公共语言运行库正在关闭或当前应用程序域正在卸载的验证值
MachineName	获取机器的 NetBIOS 名
NewLine	获取当前环境定义的换行字符串
OSVersion	获取当前平台信息
ProcessorCount	获取机器上的处理器数
StackTrace	获取文堆栈跟踪信息
System.Directory	获取系统目录的完全限定名
TickCount	获取此 DirectoryInfo 实例的名称
UserDomainName	获取系统启动后经过的毫秒数
UserInteractive	获取当前用户的关联域名
Username	获取当前进程人的名字
Version	获取描述公共语言运行库版本的 Version 对象
WorkingSet	获取映射到进程上下文的物理内存量
Exit	终止进程，退出代码

类 Environment 的主要方法信息如表 14-10 所示。

表 14-10 类 Environment 方法信息

方　　法	说　　明
ExpandEnvironmentVariables（）	将嵌入指定字符串的环境变量名替换为和该变量值等效的字符串
FailFast（）	终止进程
GetCommandLineArgs（）	返回当前进程的命令行参数的字符串数组
GetEnvironmentVariable（）	检索环境变量的值
GetEnvironmentVariables（）	检索所有环境变量的值和名
GeFolderPath（）	获取指向由指定枚举标识的系统特殊文件夹的路径
GetLogicalDrives（）	返回当前机器逻辑驱动器名和字符串数组
SetEnvironmentVariabl（）	创建、修改或删除环境变量

实例 64：使用 Environment 类输出当前系统的参数

下面将通过一个具体的实例来说明 Environment 类的具体使用流程。本实例保存在“光盘：\daima\14”文件夹内，项目名为 Environmentshiyong。实例的功能是使用 Environment 类输出当前系统的参数。实例文件的主要代码如下所示：

```
namespace Environmenthuli{
    class Program
    {
        static void Main(string[] args)
```

```
            {
                Console.WriteLine("处理器数量：" + Environment.ProcessorCount);
                Console.WriteLine("操作系统版本：" + Environment.OSVersion);
                Console.WriteLine("公共语言运行时版本：" + Environment.Version);
                Console.WriteLine("系统目录：" + Environment.SystemDirectory);
                Console.WriteLine("域名：" + Environment.UserDomainName);
                Console.WriteLine("机器名：" + Environment.MachineName);
                Console.WriteLine("用户名：" + Environment.UserName);
                Console.WriteLine("堆栈信息：" + Environment.StackTrace);
                Console.WriteLine("物理内存量：" + Environment.WorkingSet);
                Console.WriteLine("启动毫秒数：" + Environment.TickCount);
                Console.ReadKey();
            }
        }
    }
```

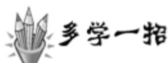
在上述实例代码中，使用了类 Environment 的方法获取当前机器的各系统参数。具体说明如下。

- 处理器数量，Environment.ProcessorCount。
- 操作系统版本，Environment.OSVersion。
- 公共语言运行时版本，Environment.Version。
- 系统目录，Environment.SystemDirectory。
- 域名，Environment.UserDomainName。
- 机器名，Environment.MachineName。
- 用户名，Environment.UserName。
- 堆栈信息，Environment.StackTrace。
- 物理内存量，Environment.WorkingSet。
- 启动毫秒数，Environment.TickCount。

上述实例代码执行后，将输出对当前程序运行环境的系统参数，如图 14-5 所示。

```
处理器数量：2
操作系统版本：Microsoft Windows NT 5.1.2600 Service Pack 2
公共语言运行时版本：2.0.50727.42
域名：HP
机器名：HP
用户名：guan
堆栈信息：   在 System.Environment.GetStackTrace(Exception e, Boolean needFileIn
fo)
   在 System.Environment.get_StackTrace()
   在 Environmenthuli.Program.Main(String[] args) 位置 E:\机械\c#\daima\14\Envir
onmenthuli\Program.cs:行号 17
   在 System.AppDomain.nExecuteAssembly(Assembly assembly, String[] args)
   在 System.AppDomain.ExecuteAssembly(String assemblyFile, Evidence assemblySec
urity, String[] args)
   在 Microsoft.VisualStudio.HostingProcess.HostProc.RunUsersAssembly()
   在 System.Threading.ThreadHelper.ThreadStart_Context(Object state)
   在 System.Threading.ExecutionContext.Run(ExecutionContext executionContext, C
ontextCallback callback, Object state)
   在 System.Threading.ThreadHelper.ThreadStart()
物理内存量：16572416
启动毫秒数：46433765
```

图 14-5　程序执行结果



类 Environment 还能够获取系统机器的特殊目录路径。例如"Program Files"、"Application Data"和"System"等。上述特殊目录由在默认情况下由系统设置，或在安装系统时进行设置。通常使用 GetFnn()方法并指定一个 SpecialFolder 枚举常数，即可检索到特殊目录的路径。SpecialFolder 枚举常数的对应特殊目录信息如表 14-11 所示。

表 14-11　SpecialFolder 枚举对应目录信息

成　　员	说　　明
ApplicationData	当前漫游用户应用程序特定数据的公共存储库
CommonApplicationData	漫游用户的配置文件
CommonProgramFiles	所有用户使用的应用程序特定数据的公共存储库
Cookies	应用程序间的共享组件目录
Desktop	网络 Cookie 的存储目录
DesktopDirectory	机器桌面上的文件对象目录
Favorites	用户收藏夹目录
History	网络的历史访问记录目录
InternetCache	网络临时文件存储目录
LocalApplicationData	非漫游用户应用程序特定数据的公共存储库
MyDocument	"我的文档"目录
MyComputer	"我的电脑"目录
MyMusic	"我的音乐"目录
MyPictures	"我的图片"目录
Personal	文件公共存储库目录
ProgramFiles	"ProgramFiles"目录
Programs	包含用户程序的目录
Recent	最近访问文件的存储目录
SendTo	包含发送菜单的目录
StartMenu	包含开始菜单的目录
Startup	用户启动程序目录
Templates	文件模板的公共存储目录
System	System 目录

14.2　C#流文件管理

通常将具有永久存储及特定顺序字节组成的有序的集合称为文件，文件和目录路径、磁盘存储和目录名有着密切的关系。流提供了一种后续的读取字节操作方式，这种后续的存储可以是各种媒介之一。例如，可以将计算机文件后续存储为网络流和内存流等。通过这些流方式的文件存储，可以提高文件的备份能力。

在.NET 框架类库中，定义了一个抽象基类 Stream，这是所有流类处理的基类。类

Stream 支持字节的读取和写入，并集成了异步支持。在 C#中的所有流类都是从 Stream 继承的，概括来说主要涉及如下三种处理类型。

- ❑ 流读取：读取传输从流到数据结构的数据。
- ❑ 流写入：写入传输从流到数据结构的数据。
- ❑ 流查找：查询和修改处理流内的当前位置。

在上述操作处理类型中，因为基础数据源或存储库的不同，而只能实现上述功能中的一部分。

在 System.IO 命名空间中，定义了专用的类对文件和流进行操作。C#中的流操作类包括 System.IO.BinaryReader、System.IO.BinaryWriter、System.IO.MarshByRefObject、System.IO.Stream、System.IO.BufferedStream、System.IO.FileStream、System.IO.MemoryStream、System.IO.TextReader、System.IO.TextWriter、System.IO.StreamReader、System.IO.StringReader、System.IO.StreamWriter、System.IO.StringWriter。

上述各个流文件操作类型的对应关系如图 14-6 所示。

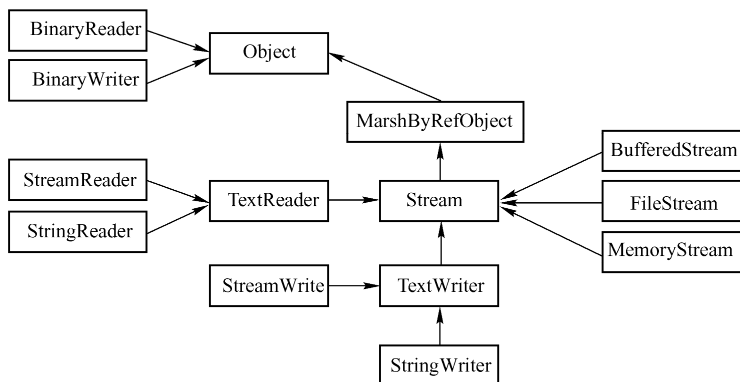


图 14-6　System.IO 命名空间流文件操作类关系图



在上述关系图中，没有使用类名的全称，即都省略了 System.IO 或 System。

14.2.1　使用 FileStream

FileStream 类是专门进行文件操作的 Stream，能够同时支持同步读写操作和异步读写操作。FileStream 类的特点是具体操作字节和字节数组，对随机文件的操作比较有效。

FileStream 类能够对文件实现低级而复杂的操作，其主要属性的信息如表 14-12 所示。

表 14-12　FileStream 类属性信息

属　　性	说　　明
CanRead	获取一个值，该值设置当前流是否支持读取
CanSeek	获取一个值，该值设置当前流是否支持查找
CanTimeout	获取一个值，该值设置当前流是否可以超时
CanWrite	获取一个值，该值设置当前流是否支持可以写入

（续）

属　性	说　明
Handle	获取当前 FileStream 所封装文件的操作系统句柄
IsAsync	获取一个值，该值设置 FileStream 是异步还是同步
Length	获取用字节表示的流长度
Name	获取传给构造函数的 FileStream 的名称
Position	获取设置当前流的位置
ReadTimeout	获取一个值，该值设置流在超时前尝试读取多长时间
SafeFileHandle	获取 SafeFileHandle 对象，该对象表示当前 FileStream 对象封装文件操作系统的句柄
WriteTimeout	获取或设置一个值，该值设置流在超时前尝试写入的时间

FileStream 类的主要方法信息如表 14−13 所示。

表 14−13　FileStream 类方法信息

方　法	说　明
BeginRead（）	开始异步读
BeginWrite（）	开始异步读写
Close（）	关闭当前流冰释放所有资源
CreateObjRef（）	创建一个对象，该对象包含生成用语与远程对象进行通信代理的所需信息
EndRead（）	等待挂起的异步读取完成
EndWrite（）	结束异步写入
Flush（）	清除流内的缓冲区
GetAccessControl（）	获取 FileSecurity 对象
Lock（）	允许在访问时并设置方式被更改
Read（）	读取流数据并计将此数据写入到缓冲区
ReadByte（）	从文件读取字节，并将读取文治提升一个字节
Seek（）	将流的当前位置设置为给定值
SetAccessControl（）	将 FileSecurity 对象描述的访问控制列表项应用于当前 FileSecurity 对象所描述的文件
SetLength（）	将流的长度设置为给定值
Synchronized（）	在指定 FileSecurity 对象周围创建线程安全包装
Unlock（）	允许其他进程访问前锁定某文件
Write（）	将从缓冲区读取的数据写入流
WriteByte（）	将一个字节写入文件流的当前位置

FileStream 类能够对文件实现各种常规操作，例如数据流的输入和输出。也能够缓冲输入和输出，从而提高了程序的性能。

FileStream 类的对象可以使用 Seek()方法来随机访问文件，在创建 FileStream 对象时需要指定如下四种信息。

1）文件名，用于设置要打开文件的路径。

2）打开模式，用于设置是否对文件执行改写、创建和打开等操作，或上述操作的集合。打开模式是由枚举 FileMode 来定义的。

3）访问模式，用于设置对文件的读写访问或读写访问的常数，是由枚举 FileAccess 来定义的。

4）共享模式，用于设置对文件操作的共享性，是由枚举 FileShare 来定义的。

FileMode 枚举中的主要成员的信息如下。

1）Append，用于打开现有文件并查找到文件尾，或创建新的文件。FileMode.Append 只能和 FileAccess.Write 一起使用。

2）Create，用于指定操作系统应该创建新文件。如果文件已经存在，则重写这个文件。如果文件不存在则使用 CreateNew，如果存在则使用 Truncate。

3）CreateNew，用于指定操作系统应该创建新文件。

4）Open，用于指定操作系统应打开此文件，如果该文件不存在则会引发异常。

5）OpenOrCreate，用于指定操作系统应打开此文件，如果不存在则创建此文件。

6）Truncate，用于指定操作系统应打开此文件，文件被打开后将被截断为 0 字节大小。

枚举 FileAccess 中的主要成员信息如下。

1）Read，用于对文件进行读取访问。

2）ReadWrite，用于对文件进行读写访问。

3）Write，用于对文件进行写访问。

下面将通过一个具体的实例来说明通过 FileStream 类实现读写处理的具体流程。

实例 65：使用 FileStream 创建指定目录的文件

本实例保存在“光盘：\daima\14”文件夹内，项目名为 useFileStream。实例文件的功能是使用 FileStream 创建指定目录的文件。实例文件 CodeFile1.cs 的主要代码如下：

```
public class useFileStream
{
    static void Main(string[] args)
    {
        byte[] m_bDataWrite = new byte[100];
        char[] nn = new char[100];

        try
        {
            //创建 E:\file.txt 的 FileStream 对象
            FileStream mm = new FileStream(@"E:\file.txt", FileMode.
            OpenOrCreate);
            nn = "My First File Operation".ToCharArray();       //将要写入
            的字符串转换成字符数组
            //通过 UTF-8 编码方法将字符数组转换成字节数组
            Encoder zz = Encoding.UTF8.GetEncoder();
            zz.GetBytes(nn, 0, nn.Length, m_bDataWrite, 0, true);
            mm.Seek(0, SeekOrigin.Begin); //设置流的当前位置为文件开始位置
            mm.Write(m_bDataWrite, 0, m_bDataWrite.Length);     //将字节数
            组中的内容写入文件
        }
        catch (IOException ex)
```

```
            {
                Console.WriteLine("出错");
                Console.WriteLine(ex.Message);
                Console.ReadLine();
                return;
            }

            Console.WriteLine("成功");
            Console.ReadLine();
            return;
        }
    }
```

上述实例代码的设计流程如下。

1）分别定义 byte 类型数组 m_bDatawrite 和 char 类型数组 nn。

2）定义 FileStream 实例 mm，用于设置新建的文件。

3）将写入文本的“你好呀”转换为字符数组。

4）对写入的文件流进行处理。

5）通过 catch 进行一场处理，如果成功则输出“成功”；如果出现异常则输出“出错”。

执行后将创建文件“E:\file.txt”，如图 14-7 所示。

成功

图 14-7　实例执行结果



多学一招

使用 FileStream 类可以实现对指定文件内容的读取。例如下面的代码:

```
    using System.IO;
    namespace duqu
    {
        public class FileStreamTest1
        {
            static void Main(string[] args)
            {
                byte[] mm = new byte[100];
                char[] nn = new char[100];
                try
                {
                    //创建 C:\111.txt 的 FileStream 对象
                    FileStream zz = new FileStream(@"C:\111.txt", FileMode.Open);
                    //设置流的当前位置为文件开始位置
                    zz.Seek(0, SeekOrigin.Begin);
                    //将文件的内容读入字节数组中
                    zz.Read(mm, 0, 100);
                }
                catch (IOException ex)
```

```
                {
                    Console.WriteLine("失败了");
                    Console.WriteLine(ex.Message);
                    Console.ReadLine();
                    return;
                }
                //通过 UTF-8 编码方法将字节数组转换成字符数组
                Decoder tt = Encoding.UTF8.GetDecoder();
                tt.GetChars(mm, 0, mm.Length, nn, 0);
                Console.WriteLine("成功了");
                Console.WriteLine(nn);
                Console.ReadLine();
                return;
            }
        }
    }
```

上述代码的设计流程如下。

1）分别定义 byte 类型数组 mm 和 char 类型数组 nn，用于数据转换。

2）定义 FileStream 实例 mm，用于设置被读取的文件名。

3）通过 catch 语句进行异常判断，如果有异常则输出对应提示。

4）定义 Decoder 实例 tt，用于获取文件内容流。

5）读取成功并输出提示。

上述代码保存在"光盘:\daima\18\duqu"，执行后如果不存在文件"C:\111.txt"，则输出对应的提示。

14.2.2 使用 StreamReader 和 StreamWriter

类 StreamReader 和类 StreamWriter 能够读写处理指定的文本文件。在下面的内容中，将详细介绍类 StreamReader 和类 StreamWriter 的具体使用方法。

1. StreamWriter

使用 FileStream 类后会十分影响效率，因为它需要将写入的文本预先转换为 UTF-8 格式。而 StreamWriter 类允许将字符和字符串直接写入文本，而不需要转换为 UTF-8 格式。StreamWriter 类的主要属性信息如表 14-14 所示。

表 14-14 StreamWriter 类属性信息

属性	说明
AutoFlush	获取一个值，该值设置 StreamWriter 是否每次写入处理后刷新缓冲区
BaseStream	获取同后备存储区连接的基础流
Encoding	获取将输入写到其中的的 Encoding
FormatProvider	获取控制格式设置的对象
NewLine	获取设置当前 TextWrite 使用的行结束符字符串

StreamWriter 类的主要方法信息如表 14-15 所示。

表 14-15　StreamWriter 类方法信息

方　　法	说　　明
Close（）	关闭当前 StreamWriter 对象基础流
Flush（）	清理编写器缓冲区，并将缓冲数据写入基础流
Synchronized（）	在指定 TextWrite 周围创建线程安全包装
Write（）	写入流数据
WriteLine（）	写入重载参数指定的数据，后面紧跟行结束符

看下面一段代码：

```
using System.IO;
namespace StreamWriterchuli{
    class Program
    {
        static void Main(string[] args)
        {
            try
    {
                //保留文件现有数据，以追加写入的方式打开 e:\123.txt 文件
                StreamWriter mm = new StreamWriter(@"e:\123.txt", true);
                //向文件写入新字符串，并关闭 StreamWriter
                mm.WriteLine("你好，这是我写入的文本!! ");
                mm.Close();
          }
          catch(IOException ex)
          {
                Console.WriteLine("异常");
                Console.WriteLine(ex.Message);
                Console.ReadLine();
                return;
          }
          Console.WriteLine("写入文本成功!");
          Console.ReadLine();
          return;
        }
    }
}
```

上述代码的设计流程如下。

1）定义 StreamWriter 实例对象 mm，设置向文件“e:\123.txt”内写入文本。

2）通过 mm.WriteLine 设置写入的文本是“你好，这是我写入的文本!!”。

3）如果有异常则通过 catch 语句输出对应提示。

4）写入成功则输出对应的“写入文本成功!”提示。

上述实例代码执行后，如果不存在文件“e:\123.txt”，则创建它，并写入文本“你好，

这是我写入的文本!!”，并输出成功提示，如图 14-8 所示。如果文件存在，则在原来基础上写入上述文本，如果执行上述程序多次，则会写入多次上述文本。

2．StreamReader

和 StreamWriter 类相比，StreamReader 类实现了另外一种读取文件数据的方法。使用 StreamReader 类的方式和 StreamWriter 类的基本相同，其主要属性信息如表 14-16 所示。

你好，这是我写入的文本！！

图 14-8　写入成功提示

表 14-16　StreamReader 类属性信息

属　　性	说　　明
BaseStream	返回基础流
CurrentEncoding	获取当前 StreamReader 对象正在使用的字符编码
EndOfStream	获取表示是否流位置在结尾的值

StreamReader 类的主要方法信息如表 14-17 所示。

表 14-17　StreamReader 类方法信息

方　　法	说　　明
Close ()	关闭当前 StreamReader 对象基础流
DiscardBufferedData ()	允许当前 StreamReader 对象丢弃当前数据
Peek ()	返回下一个可用字符
Read ()	读取流中下一个字符或一组字符
ReadBlock ()	读取流中最大 count 的字符，并从 index 开始降该数据写入 buffer
ReadLine ()	在流中读取一行字符，并将数据作为字符串返回
ReadToEnd ()	从流头到尾读取整个流
Synchronized ()	在指定 TextReader 周围创建线程安全包装

看下面一段代码：

```
using System.IO;
namespace StreamReaderchuli
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                //以绝对路径方式构造新的 StreamReader 对象
                StreamReader mm = new StreamReader(@"e:\123.txt");
                //用 ReadToEnd 方法将 e:\123.tx 中的数据全部读入字符串 nn 中，并关
                闭 StreamReader。
                string nn= mm.ReadToEnd();
                Console.WriteLine(nn);
                mm.Close();
```

```
                }
                catch (IOException ex)
                {
                    Console.WriteLine("有异常发生!");
                    Console.WriteLine(ex.Message);
                    Console.ReadLine();
                    return;
                }
                Console.ReadLine();
                return;
            }
        }
    }
```

上述代码的设计流程如下。

1）定义 StreamReader 实例对象 mm，设置读取文件“e:\123.txt”内的文本内容。

2）通过 mm.ReadToEnd 读取文件流的全部内容，并将读取值赋给 nn。

3）如果有异常则通过 catch 语句输出对应提示。

4）通过 WriteLine()将读取的数据输出显示。

执行后将读取文件“e:\123.txt”的文本内容，并输出具体的内容，如图 14−9 所示；如果文件不存在，则输出对应的异常提示。

```
你好，这是我写入的文本！！
你好，这是我写入的文本！！
你好，这是我写入的文本！！
你好，这是我写入的文本！！
你好，这是我写入的文本！！
你好，这是我写入的文本！！
你好，这是我写入的文本！！
```

图 14−9　输出被读取文件的文本内容

从上述各执行代码可以看出，当从文本文件中读、写数据时，必须首先创建一个和被操作文件名相关的 StreamReader 和 StreamWriter 实例对象。每一个 StreamReader 和 StreamWriter 对象都与一种字符编码方式相关，指定读写文本是基于哪种字符编码来进行的。在没有明确指定的情况下，StreamReader 和 StreamWriter 的默认编码是 UTF8Encoding。

14.2.3　使用 BinaryReader 和 BinaryWriter

类 BinaryReader 和类 BinaryWriter 可以读写处理指定的二进制数据。类 BinaryReader 和类 BinaryWriter 对象都是基于 FileStream 对象创建的。在读写二进制数据前，应该首先创建 FileStream 对象，然后设置文件的打开模式或访问模式，最后生成 FileStreamBinaryReader 或 BinaryWriter 对象。

1．BinaryReader

BinaryReader 类能够读取文件的二进制数据，它使用特定的编码将基元数据类型读作为二进制值。BinaryReader 类的主要属性信息如表 14−18 所示。

表 14-18 BinaryReader 属性信息

属　　性	说　　明
BaseStream	返回基础流

BinaryReader 类的主要方法信息如表 14-19 所示。

表 14-19 BinaryReader 方法信息

方　　法	说　　明
Close（）	关闭当前阅读器和基础流
PeekChar（）	返回下一个可用字符
Read（）	从当前流读取字符，但不提升字符位置
ReadBoolean（）	从当前流读取 Boolean 字符，并提升字符位置 1 个字节
ReadByte（）	从当前流读取下一字节，并提升字符位置 1 个字节
ReadBytes（）	将当前流的 count 个字节读入字节数组，并提升字符位置 count 个字节
ReadChar（）	从当前流读取下一字符，并根据 Encoding 和流中的特殊字符提升当前位置
ReadChars（）	将当前流的 count 个字符以字符数组形式返回数据，并根据 Encoding 和流中的特殊字符提升当前位置
ReadDecimal（）	从当前流读取十进制数值，并将流提升 16 个字节
ReadDouble（）	从当前流读取 8 节浮点值，并将流提升 8 个字节
ReadInt16（）	从当前流读取 2 字节有符号整数，并将流提升 2 个字节
ReadInt32（）	从当前流读取 4 字节有符号整数，并将流提升 4 个字节
ReadInt64（）	从当前流读取 8 字节有符号整数，并将流提升 8 个字节
ReadSByte（）	从当前流读取一个有符号整数，并将流提升 1 个字节
ReadSingle（）	从当前流读取 4 节浮点值，并将流提升 4 个字节
ReadString（）	从当前流读取 1 个字符串
ReadUInt16（）	使用 LittleEndian 编码从当前流中读取 2 字节无符号整数，并将流位置提升 2 个字节
ReadUInt32（）	从当前流中读取 4 字节无符号整数，并将流位置提升 4 个字节
ReadUInt64（）	从当前流中读取 8 字节无符号整数，并将流位置提升 8 个字节

看下面一段代码：

```
using System.IO;
namespace BinaryReaderchuli
{
    class Program
    {
        static void Main(string[] args)
        {
            FileStream mm = new FileStream(@"E:\123.txt", FileMode.Open,
            FileAccess.Read);
            // 通过文件流创建相应的 BinaryReader
            BinaryReader nn = new BinaryReader(mm);
            for (int i = 0; i < 11; i++)  // 从 E:\123.txt 中读取数据
            {
```

```
                    Console.WriteLine(nn.ReadByte());
                }
                nn.Close();
                mm.Close();
                Console.ReadLine();
            }
        }
    }
```

上述代码的设计流程如下。

1）定义 FileStream 实例对象 mm，设置被操作文件是“E:\123.txt”。

2）定义 BinaryReader 实例对象 nn，用于读取 mm 对象的数据。

3）通过 for 语句逐一读取文件的内容。

4）通过 ReadByte 将读取的字节输出显示。

执行后将读取文件“E:\123.txt”的文本内容，并转换为字节格式输出显示，如图 14-10 所示。

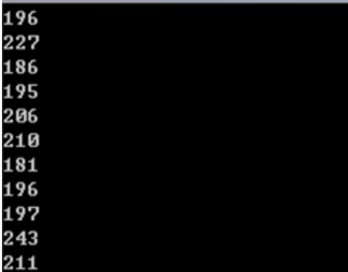


图 14-10　输出转换后的文本内容

注意

在上述执行代码中，如果文件“E:\123.txt”的文本内容有标点符号存在，则不会被转换为 ReadByte 字节格式。

2．BinaryWriter

BinaryWriter 类能够以二进制格式将数据写入到指定文件，并且可以用特定的编码将基元数据写入到字符串。使用类 BinaryWriter 的具体方法和类 BinaryReader 的类似，其主要属性信息如表 14-20 所示。

表 14-20　BinaryWriter 属性信息

属　性	说　明
BaseStream	获取 BinaryWriter 基础流

BinaryWriter 类的主要方法信息如表 14-21 所示。

表 14-21　BinaryWriter 方法信息

方　法	说　明
Close（）	关闭当前 BinaryWriter 和基础流
Flush（）	清理写入缓冲区
Seek（）	设置当前流的位置
Write（）	将值写入当前流

看下面一段代码：

```
using System.IO;
namespace BinaryWriterchuli
```

```
    {
        class Program
        {
            static void Main(string[] args)
            {
                FileStream mm = new FileStream(@"E:\123.txt", FileMode.Create);
                BinaryWriter nn = new BinaryWriter(mm);   //通过文件流创建相应的
                BinaryWriter
                for (int i = 0; i < 20; i++)
                {
                    nn.Write((int)i);                //向 E\123.txt 中写入数据
                }
                nn.Close();
                mm.Close();
                Console.WriteLine("写入成功!");
                Console.ReadLine();
            }
        }
    }
```

上述代码的设计流程如下。

1）定义 FileStream 实例对象 mm，设置被操作文件是“E:\123.txt”。

2）定义 BinaryWriter 实例对象 nn，用于向“E:\123.txt”文件内写入数据。

3）通过 for 语句设置写入的内容是 0~19 的 20 个数字。

4）通过 nn.Write 将上述数字写入，并输出成功提示。

执行后将向文件“E:\123.txt”内写入 0~19 的 20 个数字，并输出写入成功提示，如图 14−11 所示。

写入成功!

图 14−11　成功提示

14.3　XML 文件处理

XML 即可扩展标记语言，是 XtensibleMarkup Language 的缩写。XML 是一种描述数据和数据结构的语言，在当前商业领域和日常生活中得到了广泛的应用，已经日益成为数据交换的标准通用语言。在本节的内容中，将简要介绍 XML 的基本知识，并详细介绍处理 XML 文件的方法。

14.3.1　XML 简介

XML 是在 SGML 和 HTML 标记语言的基础上产生的最新语言之一，但是它并没有事先定义具体的标签，而只是提供了一个标准。任何人都可以利用这个标准来定义自己需要的新标记。

1. 文件结构

一个简单的 XML 文件标记由如下三类标记构成。

- 结构：将整个文件进行划分处理。
- 语义：将文件元素和实际事物联系起来。

❑ 样式：设置文件内元素的现实样式。

看下面一段代码：

```
<?xml version="1.0" encoding="YTF-8" standalone="yes"?>
<?xml-stylesheet type="text/xsl" href="hellostyle.xsl"?>
<message>你好，我的朋友！</message>
```

在上述代码中，将文件划分成了三行代码。其中首行是对 XML 文件的声明，代表 XML 文件的开始。XML 的声名主要包括如下三个属性。

1）version：设置当前 XML 文件符合的规范。

2）encoding：设置当前 XML 文件的编码标准。

3）standalone：设置文件是否都在这一个文件内。

第二行设置了文件的样式，设置了此 XML 文件的样式文件是 hellostyle.xsl。

第三行是自定义的处理程序信息。

2. 文件解析

需要使用 XML 解析器分析和提取 XML 文件信息，现实中常用的 XML 解析器有如下两种。

1）DOM，这是 W3C 组织推出的正式标准，它定义了一整套 API，使程序可以存取和更新 XML 文档的风格、结构和内容。当 DOM 在处理 XML 文件时，会将整个文档读入到内存中，获得文档元素的树形结构，并可以随机存取文档的内容和结构。

2）SAX，它基于事件处理，当使用 SAXAPI 来解析 XML 文档时，解析器将在文档的不同处产生事件，并提供处理每一个事件的回调方法。

3. XML 文件处理

在 C#内对 XML 文件的处理分为如下两种。

1）基于流的处理，是将 XML 作为普通的数据流来对待处理。

2）基于 DOM 的处理，是对象模型 DOM 将 XML 数据作为一组标准的对象来对待，用于处理内存中的 XML 数据。

14.3.2　基于流的处理

在.NET 的框架类库的 System.XML 命名空间内，定义了如下基于流 XML 处理的类：

System.Xml.XmlReader、System.Xml.XmlWriter、System.Xml.TextReader、System.Xml.NodeReader 和 System.Xml.TextWrite。

上述流 XML 处理类型的对应关系如图 14-12 所示。

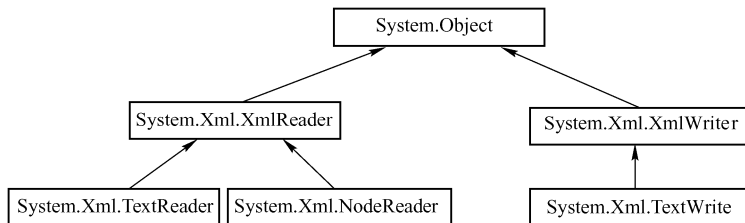


图 14-12　流 XML 处理类型的对应关系

1．使用 XmlReader

类 XmlReader 的功能是实现对 XML 数据流的只进只读访问。XmlReader 支持从流文件读取 XML 数据，其具体功能如下。

- 验证字符是否是合法的 XML 字符，验证元素名和属性名是否是有效的 XML 名称。
- 验证 XML 文档格式是否正确。
- 从 XML 流检索数据，或使用提取模型跳过不需要的纪录。

在使用 XmlReader 类时，必须先创建 XmlReader 对象，其中最为合理的是使用 Create 方法创建。Create()方法使用 XmlReaderSettings 类来指定 XmlReader 实例支持的功能，使用 XmlReaderSettings 类的属性来设置需要的功能，然后将使用 XmlReaderSettings 对象传递给方法 Create。

XmlReaderSettings 类的默认属性设置信息如表 14-22 所示。

表 14-22　XmlReaderSettings 类属性设置信息

属　　性	设置默认值
CheckCharacters	true
ConformanceLevel	ConformanceLevel.Document
IgnoreComments	false
IgnoreProcessingInstructions	false
IgnoreWhitespace	false
LineNumberOffset	0
NameTable	0
ProhibitDtd	null
Schemas	true
ValidationFlags	空的 XmlSchemaSet 对象
ValidationType	启用 ProcessIdentityConstraints
XmlResolver	新的 XmlUrlResolver

例如通过下面的代码，可以创建一个 XmlReader 对象 myXmlReader，用于处理文件“e:\123.xml”。

```
class Program
   {
      static void Main(string[] args)
      {
         string path = @"e:\123.xml";             //XML 文件路径
         //尝试读取该 XML 文件
         try
         {
            XmlReader myXmlReader = XmlReader.Create(path);
         }
         catch(Exception e)
         {
            Console.WriteLine(e.Message);
```

```
            }
            Console.ReadLine();
        }
    }
```

通过使用 XmlReader 对象，可以只对 XML 流文件进行访问，当前的节点是读取器当前所处 XML 的节点。所有的调用方法和执行的操作与当前的节点相关，所有检索到的属性反映了当前节点的值。读取器通过调用一种读取方法前进，重复调用该读取方法可以将读取器移到下一个节点。当 XmlReader 置于某个节点上之后，就可以通过 XmlReader 对象属性来获取节点的信息。通过枚举来定义节点的类型 XmlNodeType，具体的枚举常数信息如表 14-23 所示。

表 14-23 XmlNodeType 枚举节点类型信息

成员名	说明
Attribute	Attribute 节点可以具有 Text 和 EntityReference 子节点类型，Attribute 节点不以任何其他节点类型子节点的形式出现，不能将其视为 Element 的子节点
CDATA	CDATA 节用于转义否则将被识别为标记的文本块，CDATA 节点不能有任何子节点，它可以以 DocumentFragment、EntityReference 和 Element 节点子级的形式出现
Comment	Comment 节点不能有任何子节点，它可以以 Document、DocumentFragment、Element 和 Entity Reference 节点子级的形式出现
Document	作为文档树的根的文档对象提供对整个 XML 文档的访问，Document 节点可以具有以下子节点类型：XmlDeclaration、Element（最多一个）、ProcessingInstruction、Comment 和 DocumentType 它不能以任何节点类型子级的形式出现
DocumentFragment	表示文档片段。DocumentFragment 节点将节点或子树与文档关联起来，而实际上并不包含在该文档中，DocumentFragment 节点可以具有以下子节点类型：Element、ProcessingInstruction、Comment、Text、CDATA 和 EntityReference，它不能以任何节点类型子级的形式出现
DocumentType	DocumentType 节点可以具有以下子节点类型：Notation 和 Entity。它可以以 Document 节点子级的形式出现
Element	表示元素，例如“<item>” Element 节点可以具有以下子节点类型：Element、Text、Comment、ProcessingInstruction 、 CDATA 和 EntityReference 。它可以是 Document 、DocumentFragmentEntityReference 和 Element 节点的子级
EndElement	是结束元素标记，当 XmlReader 到达元素的末尾时返回
EndEntiy	由于调用 ResolveEntity 而使 XmlReader 到达实体替换的末尾时返回
Entity	表示实体声明，Entity 节点可以具有表示已展开实体的子节点（例如 Text 和 EntityReference 节点），它可以以 DocumentType 节点子级的形式出现
EndEntiyReference	表示对实体的引用，EntityReference 节点可以具有以下子节点类型：Element、ProcessingInstruction、Comment、Text、CDATA 和 EntityReference，它可以以 Attribute、DocumentFragment、Element 和 EntityReference 节点子级的形式出现
None	如果未调用 Read 方法，则由 XmlReader 返回
Notation	这是文档类型声明中的表示法，Notation 节点不能具有任何子节点，它可以以 DocumentType 节点子级的形式出现
ProcessingInstruction	表示处理指令，ProcessingInstruction 节点不能具有任何子节点，它可以以 Document、DocumentFragment、Element 和 EntityReference 节点子级的形式出现
SignificantWhitespace	混合内容模型中标记间的空白或 xml:space="preserve"范围内的空白
Text	表示节点的文本内容，Text 节点不能具有任何子节点，它可以以 Attribute、DocumentFragment、Element 和 EntityReference 节点的子节点的形式出现
Whitespace	标记间的空白
XmlDeclaration	表示 XML 声明，XmlDeclaration 节点必须是文档中的第一个节点，它不能具有子级，它是 Document 节点的子级，它可以具有提供版本和编码信息的属性

属性 Value 可以获取当前节点的文本内容，其返回值取决于当前节点的类型，常用节点类型的对应值信息如表 14-24 所示。

表 14-24　节点类型的对应值信息

节 点 类 型	值
Attribute	属性的值
CDATA	CDATA 节的内容
Comment	注释的内容
DocumentType	内部子集
ProcessingInstruction	除指令目标外的全部内容
SignificantWhitespace	混合内容模型中任何标记间的空白
Text	文本节点的内容
Whitespace	标记间的空白
XmlDeclaration	声明的内容
除上述外的其他类型	空字符串

2．使用 XmlWriter

类 XmlWriter 的功能是，以只进、只写、非缓存的方式来生成 XML 流，这是一个抽象的基类。XmlWriter 类的具体功能如下。

- 验证字符是否是合法的 XML 字符，验证元素名和属性名是否是有效的 XML 名称。
- 验证 XML 文档格式是否正确。
- 将二进制字节编码为 base64 或 binhex，并写出生成的文本。
- 使用公共语言运行库类型传递值，而不使用字符串。
- 将多个文档写入一个输出流。
- 写出有效的名称、限定名和名称标记。

和 XmlReader 类一样，使用 Create()方法创建 XmlWriter 对象是最为合理的使用方式。Create()方法是一个静态方法，可以使用 XmlRWriterSettings 类的属性来实现文件的写入操作。

XmlWriterSettings 类的默认属性的设置信息如表 14-25 所示。

表 14-25　XmlWriterSettings 类属性设置信息

属　　性	设置默认值
CheckCharacters	true
CloseOutput	false
ConformanceLevel	ConformanceLevel.Document
Encoding	Encoding.UTF-8
Indent	false
IndentChars	2 个空格
NewLineChars	\r\n
NewLineHanding	NewHanding.Replace
NewLineOnAttributes	false
OmitXmlDeclaration	false

例如，通过下面的代码可以创建一个名为“123.xml”的文件：

```
using System.Xml;
namespace fuzhi
{
    class Program
    {
        static void Main(string[] args)
        {
        //XML 文件路径
        string mm = @"e:\new.xml";
        string nn = @"e:\123.xml";
        //尝试读取该 XML 文件
        try
        {
                //创建设置
                XmlWriterSettings mySettings = new XmlWriterSettings();
                mySettings.Indent = true;
                mySettings.IndentChars = ("    ");
                // 输入 XML 数据
                XmlReader myReader=XmlReader.Create(nn);
                myReader.ReadToDescendant("fuzhi");
                //创建 XmlWriter 的实例
                XmlWriter myWriter=XmlWriter.Create(mm);
                myWriter WriteNode("myReader,true");
                myWriter.Flush();
         }
         catch (Exception e)
         {
                Console.WriteLine(e.Message);
         }
         Console.ReadLine();
      }
    }
}
```

上述代码的设计流程如下：

1）定义 string 类型变量 path，设置创建的文件名为“E:\123.xml”。

2）定义 XmlWriterSetting 对象 mySettings。

3）设置对象 Indent 属性值为 true。

4）通过 WriteStartElement()分别写入 XML 文件的元素。

执行后将按照指定格式创建一个名为“123.xml”的文件，如图 14-13 所示。

使用 WriteNode()方法，可以复制在 XmlReader 或 XPathNavagator 对象的当前位置上发现的整个元素的节点。当调用 WriteNode()方法时，会将源对象内的所有内容复制到 XmlWriter 实例中。

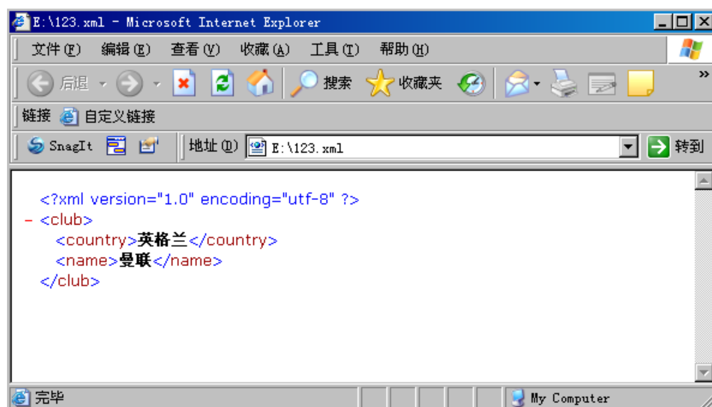


图 14-13　创建的文件

假如有一个 XML 源文件“123.xml”，具体代码如下：

```
<?xmlversion="1.0"encoding="utf-8"?>
<fuzhidate="01-01-08">
<name>zhang</name>
<nation>china</nation>
</fuzhi>
```

可以使用下面的代码将上述文件内容复制到“e:\new.xml”中。

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Xml;
namespace chuangjian
{
    class Program
    {
        static void Main(string[] args)
        {
            string path = @"e:\123.xml";            //XML 文件路径

            //尝试读取该 XML 文件
            try
            {
                //创建设置
                XmlWriterSettings mySettings = new XmlWriterSettings();
                mySettings.Indent = true;
                mySettings.IndentChars = ("    ");

                XmlWriter myWriter = XmlWriter.Create(path, mySettings);
                //创建 XmlWriter 的实例
```

```
                // 输入 XML 数据
                myWriter.WriteStartElement("club");
                myWriter.WriteElementString("country", "英格兰");
                myWriter.WriteElementString("name", "曼联");
                myWriter.WriteEndElement();
                myWriter.Flush();

            }
            catch (Exception e)
            {
                Console.WriteLine(e.Message);
            }

            Console.ReadLine();
        }
    }
}
```

上述代码的设计流程如下：

1）定义 string 类型变量 path 和 nn，其中 nn 是被复制的源 XML 文件，而 path 是复制 nn 内容后创建的文件。

2）定义 XmlWriterSetting 对象 mySettings。

3）设置 ReadToDescendant 参数为“fuzhi”。

4）调用 WriteNode()方法来复制文件内容。

执行后将复制文件“e:\123.xml”的内容到“e:\new.xml”，如图 14-14 所示。

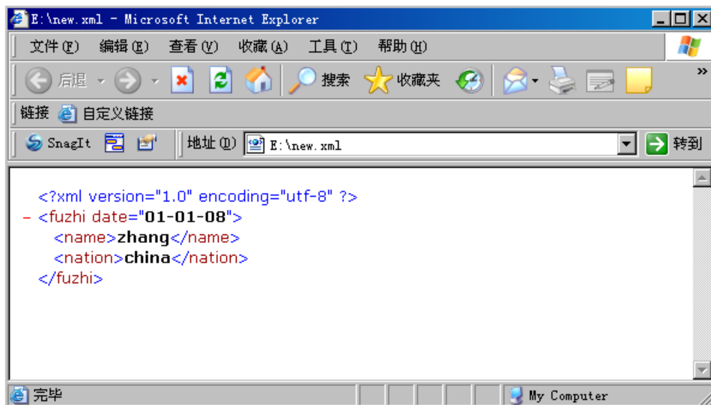


图 14-14　复制创建后的新文件

实例 66：使用 XmlWriter 类编写 XML 文件

下面将通过本实例来讲解使用 XmlWriter 类编写 XML 文件的具体方法，本实例保存在“光盘:daima\14”文件夹内，项目名为 writeread。实例文件的功能是，在项目的“bin\Debug”目录下生成指定格式的文件“456.xml”。实例文件的主要代码如下：

```
namespace writeread {
    class Program
    {
    public static void Main(string[] args)
    {
        string path = "456.xml";
        if (path.Length == 0)
        {
            Console.WriteLine("请指定 XML 文件名。");
            return;
        }
        XmlWriter writer = null;
        try
        {
            // 创建 XmlWriterSettings 对象
            XmlWriterSettings mm = new XmlWriterSettings();
            // 设置 XmlWriterSettings 对象的属性
            mm.Indent = true;
            mm.IndentChars = ("");
            // 使用 XmlReaderSettings 对象来创建 XmlReader 对象
            writer = XmlWriter.Create(path, mm);
            WriteXml(writer);
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
        }
        finally
        {
            if (writer != null)
                writer.Close();
        }
        Console.ReadKey();
    }
    private static void WriteXml(XmlWriter writer)
    {
        try
        {
            writer.WriteStartElement("qiudui");
            WriteChildNode(writer, "AA", "英国", "联赛第一");
            WriteChildNode(writer, "BB", "西班牙", "联赛第二");
            WriteChildNode(writer, "CC", "意大利", "联赛第二");
            WriteChildNode(writer, "DD", "德国", "联赛第四");
            writer.WriteEndElement();
        }
        catch (XmlException ex)
```

```
                {
                    Console.WriteLine(ex.Message);
                }
            }
            private static void WriteChildNode(XmlWriter writer,
                            string title, string author, string price)
            {
                writer.WriteStartElement("club");
                writer.WriteStartElement("name");
                writer.WriteValue(title);
                writer.WriteEndElement();
                writer.WriteStartElement("country");
                writer.WriteValue(author);
                writer.WriteEndElement();
                writer.WriteStartElement("paiming");
                writer.WriteValue(price);
                writer.WriteEndElement();
                writer.WriteEndElement();
            }
        }
    }
```

上述代码的设计流程如下。

1）定义 string 类型变量 path，设置创建的 xml 文件名为“456.xml”。

2）验证被创建文件名的长度，如果非法则输出提示。

3）定义 XmlReaderSettings 对象 mm。

4）设置 XmlReaderSettings 对象属性 Indent 的值为 true。

5）使用 XmlReaderSettings 对象来创建 XmlReader 对象 writer。

6）定义方法 WriteXml()，将指定的信息写入到 XML 文件内。

7）定义方法 WriteChildNode()，将指定的 XML 元素写入到创建的文件内。

执行后将在“bin\Debug”目录下生成指定的文件“456.xml”，如图 14-15 所示。

```
<?xml version="1.0" encoding="utf-8" '
- <qiudui>
  - <club>
      <name>AA</name>
      <country>英国</country>
      <paiming>联赛第一</paiming>
    </club>
  - <club>
      <name>BB</name>
      <country>西班牙</country>
      <paiming>联赛第二</paiming>
    </club>
  - <club>
      <name>CC</name>
      <country>意大利</country>
      <paiming>联赛第三</paiming>
    </club>
  - <club>
      <name>DD</name>
      <country>德国</country>
      <paiming>联赛第四</paiming>
    </club>
  </qiudui>
```

图 14-15 生成的文件



多学一招

也可以通过使用 XmlWriter 类编写并生成 XML 文件。具体代码如下:

```
    usingSystem.Xml;
    namespace duqu
```

```
{
class Program
{
public static void Main(string[]args)
{
string path="456.xml";
if(path.Length==0)
{
Console.WriteLine("请指定XML文件名。");
return;
}
XmlWriterwriter=null;
try
{
//创建XmlWriterSettings对象
XmlWriterSettingsmm=newXmlWriterSettings();
//设置XmlWriterSettings对象的属性
mm.Indent=true;
mm.IndentChars=("");
//使用XmlReaderSettings对象来创建XmlReader对象
writer=XmlWriter.Create(path,mm);
WriteXml(writer);
}
catch(Exceptionex)
{
Console.WriteLine(ex.Message);
}
finally
{
if(writer!=null)
writer.Close();
}
Console.ReadKey();
}
private static void WriteXml(XmlWriterwriter)
{
try
{
writer.WriteStartElement("qiudui");
WriteChildNode(writer,"张三","建筑工人","工地打工");
WriteChildNode(writer,"李四","工人","工厂上班");
WriteChildNode(writer,"王五","知识分子","在办公室");
writer.WriteEndElement();
}
catch(XmlExceptionex)
{
```

```
Console.WriteLine(ex.Message);
}
}
private static void WriteChildNode(XmlWriterwriter,
stringtitle,stringauthor,stringprice)
{
writer.WriteStartElement("club");
writer.WriteStartElement("name");
writer.WriteValue(title);
writer.WriteEndElement();
writer.WriteStartElement("country");
writer.WriteValue(author);
writer.WriteEndElement();
writer.WriteStartElement("paiming");
writer.WriteValue(price);
writer.WriteEndElement();
writer.WriteEndElement();
}
}
}
```

上述代码的设计流程如下。

1）定义 string 类型变量 path，设置创建的 xml 文件名为“E:\456.xml”。

2）验证被创建文件名的长度，如果非法则输出提示。

3）定义 XmlReaderSettings 对象 mm。

4）设置 XmlReaderSettings 对象属性 Indent 的值为 true。

5）使用 XmlReaderSettings 对象来创建 XmlReader 对象 writer。

6）定义方法 WriteXml()，将指定的信息写入到 XML 文件内。

7）定义方法 WriteChildNode()，将指定的 XML 元素写入到创建的文件内。

将上述代码保存为“光盘:\daima\19\duqu”，执行后将在“bin\Debug”目录下生成指定文件“456.xml”，如图 14−16 所示。




```
<?xml version="1.0" encoding="utf-8"?>
<qiudui>
    <club>
        <name>张三</name>
        <country>建筑工人</country>
        <paiming>工地打工</paiming>
    </club>
    <club>
        <name>李四</name>
        <country>工人</country>
        <paiming>工厂上班</paiming>
    </club>
    <club>
        <name>王五</name>
        <country>知识分子</country>
        <paiming>在办公室</paiming>
    </club>
</qiudui>
```

图 14−16　生成的文件

14.3.3 基于 DOM 的处理

DOM 是文档对象模型，能够将 XML 作为一组标准的对象来进行处理。在 System.Xml 命名空间内，提供了 XML 文档、片断、节点或节点集的编程表示方式。当 DOM 处理 XML 文件时，XML 文件的所有数据都会被放在内存中。在.NET 框架类库中，定义了一系列处理 DOM 的类，其中最为常用的是 XmlNode 和 XmlDocument。

1．XmlNode 和 XmlDocunment 基础

XmlDocunment 类使用方法 Load()将文档放入内存中。在方法 Load 中，包含了可以从不同格式中获取数据的重载方法。通过类 XmlDocunment 中的方法 LoadXml()，可以从字符串中读取 XML。假设有一个名为 123.xml 的文件，具体代码如下：

```
<?xmlversion="1.0"encoding="utf-8"?>
<qiudui>
<club>
<name>AA</name>
<country>英国</country>
<paiming>联赛第一</paiming>
</club>
<club>
<name>BB</name>
<country>西班牙</country>
<paiming>联赛第二</paiming>
</club>
<club>
<name>CC</name>
<country>意大利</country>
<paiming>联赛第二</paiming>
</club>
<club>
<name>DD</name>
<country>德国</country>
<paiming>联赛第四</paiming>
</club>
</qiudui>
```

可以使用下面的代码将上述 XML 文件的内容全部读入到内存中：

```
XmlDocunmentmm=newXmlDocument();
mm.Load(“123.xml”);
```

在类 XmlDocunment 中，提供了查看和处理整个 XML 文档节点的方法。并且 XmlDocunment 和 XmlNode 实现了互动，提高了处理的效率。XmlDocunment 和 XmlNode 可以通过方法和属性进行如下操作处理：

- ❑ 访问和修改 DOM 的特定节点。
- ❑ 检索除节点所包含信息外的所有节点。

当将 XML 读入内存时会自动调用节点，因为并不是所有的节点都是同一个类型，所以 XML 元素具有不同处理指令的规则和语法。为此，在读取各种数据时，将为每个节点分配一种节点类型，这个节点类型能够确定节点的特性和功能。节点的类型是由 XmlNodeType 所定义的。

2．使用 ImportNode 方法

XmlDocunment 类中 ImportNode 方法的功能是，将节点从一个 XmlDocunment 对象复制到另一个 XmlDocunment 对象中。复制之后原文件的内容不会改变，在新文件中会增加一个新节点对象。

实例 67：使用 ImportNode 方法实现复制指定 XML 文件

下面通过一个具体的实例来说明使用 ImportNode()方法复制指定 XML 文件的过程。本实例保存在“光盘：\daima\14”文件夹内，项目名为 copychuli。实例文件的功能是将指定节点的内容复制到文件“123.xml”中。实例文件的主要代码如下所示：

```
namespace copychuli
{
    class Program
    {
        public static void Main()
        {
            try
            {
                XmlDocument mm = new XmlDocument();
                mm.LoadXml("<club><name>AA</name><paiming format= \"RMB\">
1</paiming></club>");
                XmlDocument nn = new XmlDocument();
                nn.LoadXml("<country>英格兰</country>");
                XmlNode newNode = mm.ImportNode(nn.DocumentElement, true);
                mm.DocumentElement.AppendChild(newNode);
                mm.Save("123.xml");
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
        }
    }
}
```

上述代码的设计流程如下。

1）定义 XmlDocument 对象 mm。

2）然后使用 LoadXml 方法设置导入的节点信息。

3）定义 XmlDocument 对象属性 nn，然后使用 LoadXml()方法设置导入的节点信息。

4）定义 XmlNode 对象 newNode，通过 AppendChild()方法将导入元素添加到 XML 中。

5）通过调用对象 mm 的 Save()方法，将导入数据存放在文件“123.xml”内。

执行后将把指定的节点元素内容复制到文件“123.xml”内，如图 14-17 所示。

```
- <club>
    <name>A A </name>
    <paiming format="RMB">1</paiming>
    <country>英格兰</country>
  </club>
```

图 14-17　处理后的文件“123.xml”

当把 XML 文档放入到内存后，可以对节点进行删除处理。或删除节点类型中的内容和值。在删除某个节点时，会同时删除这个节点的子节点。如果要删除文档内的多个节点，则可以使用 RemoveAll()方法来删除节点和属性。

多学一招

C#中一般采用如下两种方法来删除节点的属性。

（1）从属性集合中删除

从属性集合中删除属性的具体步骤如下:

第一步，使用 XmlAttributeCollectionmm=elem.Attributes 获取各元素的属性集合。

第二步，使用 Remove()删除指定的属性；使用 RemoveAll()删除集合中所有的属性，并保留没有属性的元素；使用 RemoveAt()删除指定索引号的属性。

（2）获取属性删除

获取属性删除是指，获取某元素或某集合中的属性，然后将此属性的节点删除。当使用 RemoveAll()方法来删除节点和属性时，被它操作的节点属性值不能为空值，即使为空也要使赋值为 Empty。如果要删除从 XmlCharacterData 继承的节点类型，可以使用 DeleteData()方法来实现。DeleteData()方法可以在节点内删除某范围内的字符。

14.4　疑难问题解析

本章详细介绍了 C#文件管理的基本知识。本节将对本章中比较难以理解的问题进行讲解。

读者疑问：怎样修改某个节点的值？

解答：如果需要修改某个节点的值，则可以通过如下五种方式实现:

- 使用 Value 属性更改节点的值。
- 使用新节点来替换原节点。
- 使用 RemoveChild 将现有的节点替换为新的节点。
- 使用 AppendData、InsertData 或 ReplaceData 从 XmlCharacterData 类继承的节点添加附加字符。
- 在从 XmlCharacterData 类继承的节点类型上，使用 DeleteData 方法删除某范围的字符，实现内容修改。

读者疑问：XML 节点具有什么特点？

解答：XmlNode 的对象具有自身的方法和属性，用来实现对 XML 节点数据的定义。XML 节点具有如下三个主要特点：

- 每个节点都有一个父节点，XmlDocunment 对象除外。
- 大多数节点可以有多个子节点，子节点是和节点相邻的下一级节点。节点的所有子节点包含在 ChildNodes 属性中。
- 同级别的节点是兄弟节点。

因为属性属于元素，所以要获取属性就必须首先获取它所属的元素。通过元素节点的 HasAttribute 方法能够判断元素是否存在特定的属性。使用方法 GetAttribute 可以获取指定的属性。



职场点拨——程序员的几个坏习惯

良好的编程习惯是每个程序员都应该具备的工作素质，但是有一些程序员的身上总有这样或者那样的坏习惯，这些坏习惯在一些从业时间不是很长的程序员身上表现得特别突出。总结起来，有如下六个常见的坏习惯。

（1）缺乏规划和需求分析

很多程序员，当接到一个任务的时候。没有做需求分析和详细设计，一开始就打开编程工具定义了一堆的模块，然后就在这些模块的基础上进行工作，结果整个程序就像胡乱搭起来的棚子，随着代码的增多，自己也糊涂了。即使完成了代码编写，表面看来也实现了所要的功能，但是这样的程序往往是逻辑混乱的，不仅理解困难，维护起来也十分的困难。

（2）随意填代码

有些程序员，在代码维护或编写程序时，硬往程序中添加不必要的或者与对象和逻辑无关的代码。有些程序员为了偷懒，不愿意从下层将代码维护好，而是从上层开始修改。例如程序员 AA 需要做一个报表程序，他将 SQL 语句直接在界面层进行调用，这样就破坏了系统的层次和封装。并且他还在不相干的对象中添加操作其他对象的代码。

（3）盲目使用新技术

很多程序员喜欢炫耀自己的优秀，每当刚刚学会一门新技术，就忙不迭的用到项目中去，而不管这种技术是否适用于当前的程序。应用新的技术的出发点是好的，但是也要看时机和场合。

（4）糊窗操作

如果在程序中出现 bug 就如同窗户破了一个洞，有的人不是从源头上进行解决，不去寻找合理的解决方案，而是就地将这个漏洞堵上。其后果可能是真的堵上了，也可能因为其改动带来更多的问题，也可能是堵住了一个地方，回头发现又有其他的漏洞，然后不停地往程序上打补丁。

（5）很长的函数代码

函数是为了实现一个功能而编写的，函数严格体现了面向对象技术的优越性。所以无论

多么复杂的功能，尽量用最合理的函数实现。这里的合理函数要求具有代码不能太长的特质，笔者曾发现有的人一个函数居然超过 1000 行，这样的代码很恐怖，维护起来很浪费效率，并且过长的代码会影响视觉。

（6）随意修改代码

很多程序员自以为自己的水平高，所以在维护旧代码的过程中，喜欢修改别人的代码。殊不知每修改一个程序就留下一些隐患，最终结果是会将一个原本好好的代码变得到处都是隐患。所以应该尽量少修改代码。

第 15 章　Windows 窗体编程

因为 Windows 环境下的多数应用程序都是基于窗体的，所以窗体应用是十分重要的编程应用模块。在前面的内容中，讲解的都是基于命令行的控制台应用程序。从本章的内容开始，将带领读者开始学习 Windows 窗体编程知识。在本章的内容中，将对 C#窗体编程的基本知识进行简要介绍，并通过具体的实例来介绍各窗体控件的具体使用流程。通过本章能学到如下知识。

- 创建和使用窗体。
- 使用控件和组件。
- 使用窗体菜单。
- 工具栏和窗体对话框。
- 职场点拨——初次见面要留下好印象。

2008 年 XX 月 XX 日，暴雨

雨哗哗的下，我的心思全跑到客户身上了。因为明天我要和一个重要的客户见面，这是第一次见对方，为了给对方留个好印象，我得好好琢磨下穿什么衣服。



一问一答

小菜：累死我了，我终于选好明天见客户要穿的衣服了！

Wisdom：嗯，见客户当然得穿着得体，这是门面，这样才能给对方留下好的印象！在本章的最后，我将介绍一些给客户留下好印象的方法。

小菜：见客户需要注意形象，我们设计 C#程序时，也需要注意门面吗？

Wisdom：当然了，在 C#领域中，一个项目也要考虑到门面的设计，一个好的外观当然会带来客户的好感。一个赏心悦目的软件界面会给软件使用者带来愉快的心情。

小菜：C#中什么能体现出界面的美观？

Wisdom：在桌面应用中，窗体程序和 Web 最能体现出界面的美观，本章将介绍开发窗体程序的基本知识。介绍菜单、工具栏和对话框的基本知识和操作流程，逐步引导你进入 C#高级的世界。

15.1　窗体编程基础

在使用 C#进行应用程序开发时，最基本的开发工具就是前面介绍的 Visual Studio2010。使用 Visual Studio 2010 的窗体设计器，可以方便地实现窗体开发。在本节的内容中，将对

C#窗体编程的基本知识进行简要介绍。

15.1.1 窗体概述

Windows 窗体是框架技术的智能客户端技术，通过窗体可以实现信息显示、信息输入和远程通信等功能。在 Windows 应用中，窗体是向用户展示信息的视图界面。通常窗体通过标题、控制框、边框等元素实现窗体展示和控制。

在编程应用中，窗体被看做是一个对象。窗体的类定义了生成窗体的模板，在 2010 类库的 System.Windows.Forms 命名空间内定义的 Form 类是所有窗体类的基类。

窗体通过本身包含的控件来显示数据或接受用户的输入数据，控件只能被包含在窗体内，而不能在 Windows 里独立显示。当使用鼠标或键盘按键来操作窗体或窗体控件时，将会生成事件。应用程序代码会对上述事件进行响应，并调用程序来处理这些事件，这样就实现了和用户间的信息交互。

为了提高 Windows 窗体编程的应用范围，在 System.Windows.Forms 命名空间内还定义了许多个和窗体、控件相关的组件。通过对这些组件的调用，可以高性能的实现某特定功能。

在编写窗体程序时，应该首先设计窗体的外观效果，并且在其中插入需要的控件和组件。上述操作可以使用 Visual Studio 2010 轻松实现，Visual Studio 2010 提供了一个图形化的可视化窗体设计器，开发人员可以使用其所见即所得的效果实现项目开发。Visual Studio 2010 图形化的可视化窗体设计器界面如图 15-1 所示。

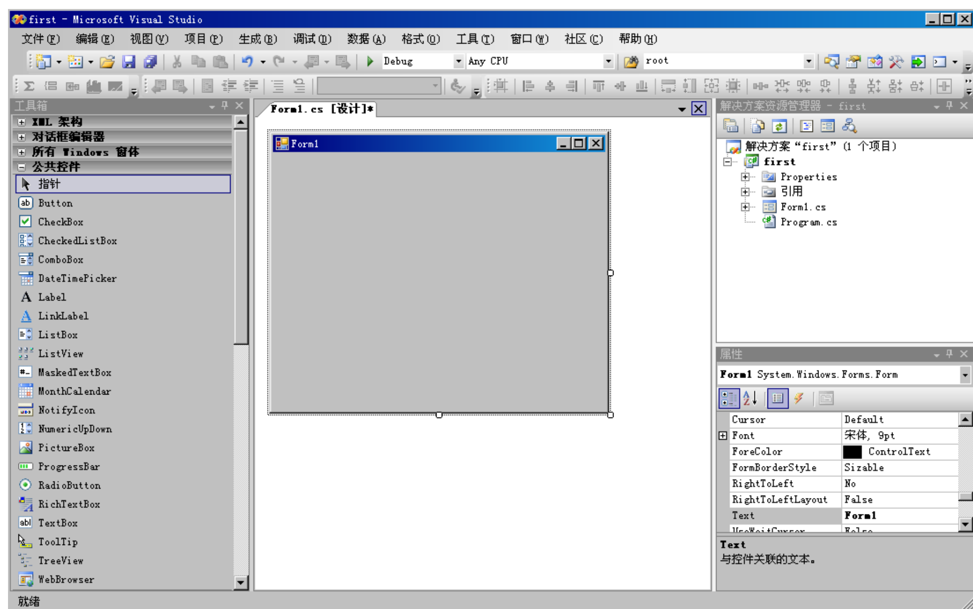


图 15-1 Visual Studio.NET 2010 图形化的可视化窗体设计器界面

通过 Visual Studio.NET 2010 的窗体设计器可以方便地实现窗体程序开发，并且利用 Visual Studio 2010 的“工具箱”可以将控件拖动或绘制到窗体内。Visual Studio 2010 窗体设计器的“工具箱”主要包括如下工具：所有 Windows 窗体、公共控件、容器、菜单和工具

栏、数据、组件、打印、对话框等。

15.1.2　一个简单的窗体程序

为了加深读者对窗体编程的理解，下面将通过一个具体的实例来说明窗体程序的创建过程。

实例 68：创建窗体程序

本实例保存在“光盘：\daima\15”文件夹内，项目名为 first。本实例的具体实现流程如下：

1）打开 Visual Studio 2010，依次选择“文件”|“新建”|“项目”命令，弹出“新建项目”对话框，如图 15-2 所示。

2）在“模板”选项中选择“Windows 应用程序”，在名称中输入项目名“first”，具体如图 15-3 所示。

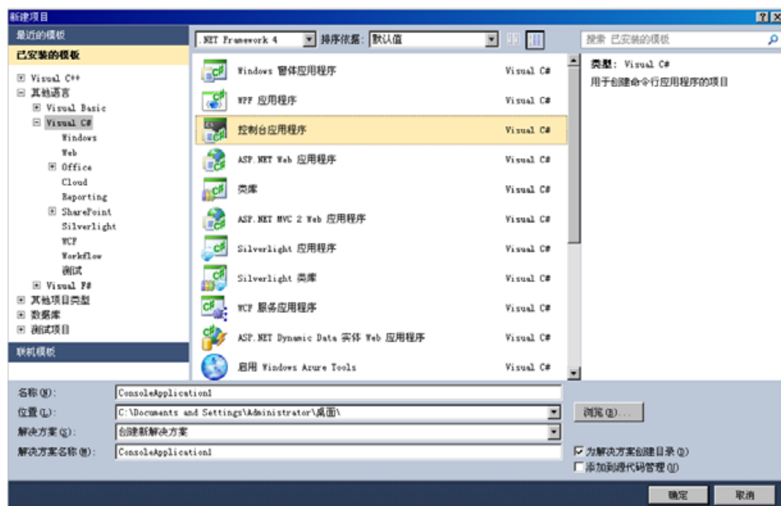


图 15-2　“新建项目”对话框

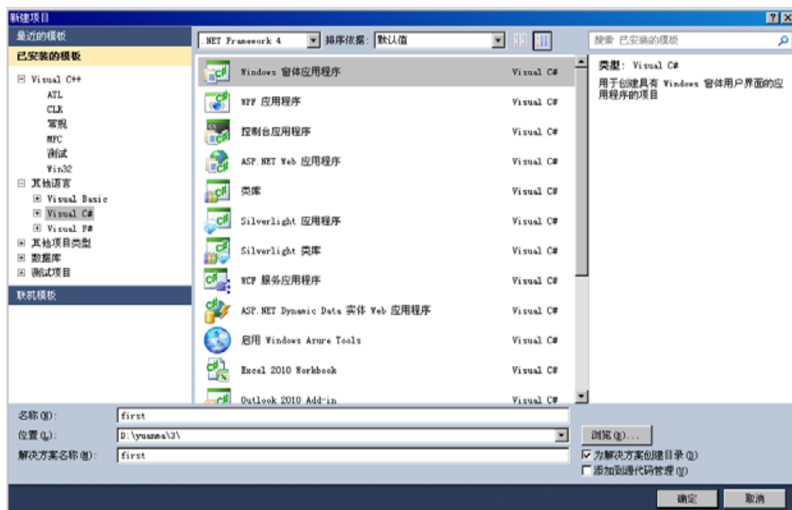


图 15-3　创建项目

3）单击“确定”按钮后将在 Visual Studio 2010 界面生成一个 Windows 窗体的应用程序项目，并自动生成一个名为 form1 的窗体，如图 15-4 所示。

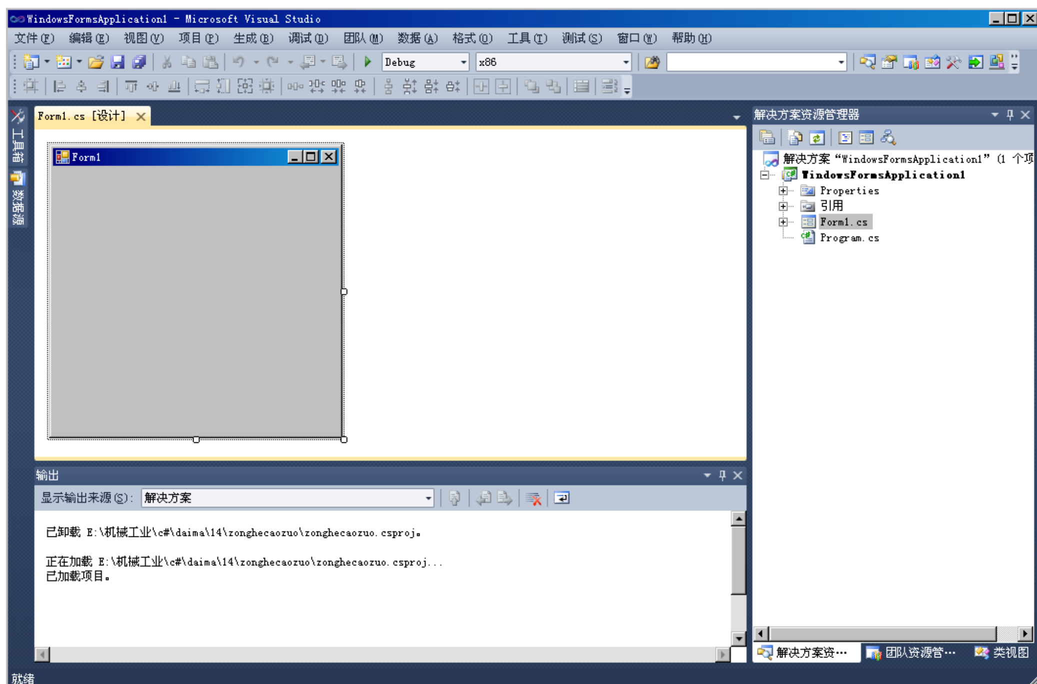


图 15-4　生成窗体项目

4）选中图 15-4 中间的 form1 窗体，然后来到右侧的“属性”窗口，设置“Text”属性为“how are you”，如图 15-5 所示。

5）单击顶部的运行按钮“▶”，将会生成一个 Windows 窗体程序，如图 15-6 所示。

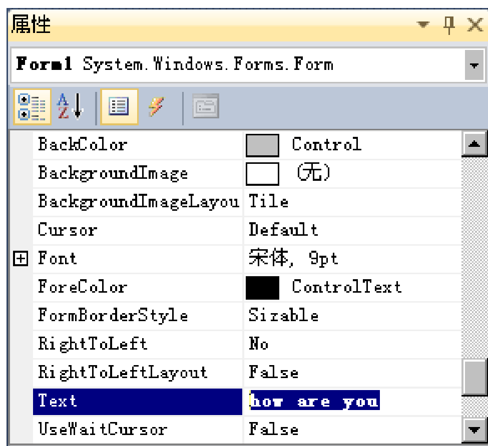


图 15-5　设置属性



图 15-6　生成窗体程序的运行结果

经过上述操作后，就成功实现了一个简单窗体程序的效果。在上述实现过程中，Visual Studio 2010 自动的生成了 C#窗体代码文件 Form1.cs。文件 Form1.cs 的具体源代码如下：

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
namespace first{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
        private void Form1_Load(object sender, EventArgs e)
        {
        }
    }
}
```

从上述实例的实现过程可以看出，使用 Visual Studio 2010 能够开发 Windows 窗体程序，并且使用的是可视化的图形操作界面，所以比较容易上手。

其中，在 Visual Studio 2010“解决方案资源管理器”Form1.cs 下有如下两个次级节点文件。

1. Form1.Designer.cs

在文件 Form1.Designer.cs 中，包含了当前窗体中所有的控件声明和 InitializeComponent()方法。方法 InitializeComponent()用于初始化窗体和窗体内的控件，并设置了它们的属性。具体代码如下：

```
namespace first{
    partial class Form1
    {
        /// <summary>
        /// 必需的设计器变量。
        /// </summary>
        private System.ComponentModel.IContainer components = null;
        /// <summary>
        /// 清理所有正在使用的资源。
        /// </summary>
        /// <param name="disposing">如果应释放托管资源，为 true；否则为
        false。</param>
        protected override void Dispose(bool disposing)
        {
            if (disposing && (components != null))
            {
                components.Dispose();
            }
```

```
            base.Dispose(disposing);
        }
        #region Windows 窗体设计器生成的代码
        /// <summary>
        /// 设计器支持所需的方法 - 不要
        /// 使用代码编辑器修改此方法的内容。
        /// </summary>
        private void InitializeComponent()
        {
            this.SuspendLayout();
            this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 12F);
            this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
            this.ClientSize = new System.Drawing.Size(355, 273);
            this.Name = "Form1";
            this.Text = "how are you";
            this.Load += new System.EventHandler(this.Form1_Load);
            this.ResumeLayout(false);
        }
        #endregion
    }
}
```

当在 Visual Studio 2010 可视界面修改窗体的属性时，上述属性的值会随之改变。

2．Form1.resx

文件 Form1.resx 是 XML 格式的资源文件，包含了当前窗体的所有资源信息，例如字符串、图标和位图等信息。上述实例的主程序是自动生成的文件 Program.cs，它定义了当前项目的入口函数 Main()。具体代码如下：

```
using System;
using System.Collections.Generic;
using System.Windows.Forms;
namespace first{
    static class Program
    {
        /// <summary>
        /// 应用程序的主入口点
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new Form1());
        }
    }
}
```

在上述入口程序内，通过 using 指令引用了 System.Windows.Forms 命名空间，实现了对 System.Windows.Forms.dll 程序集内类的引用。

15.2 创建和使用窗体

编写窗体程序是 C#的重要应用之一，通过窗体程序可以编写需要的 Windows 程序。在本节的内容中，将对 C#窗体程序的创建和使用进行简要介绍。

15.2.1 创建窗体

System.Windows.Forms 命名空间中的 Form 类是所有窗体的基类。使用 Visual Studio 2010 来创建窗体的基本流程如下。

1）创建一个新的“Windows 应用程序”项目。

2）从 System.Windows.Forms.Form 类派生一个子类，并在子类内定义窗体的特征。

3）通过“解决方案管理器”可以实现对项目程序的管理。具体说明如下：

- ❑ 在项目名称上单击鼠标右键，在弹出选项中依次选择“添加”|“Windows 窗体”命令，在项目内添加一个窗体，如图 15-7 所示。
- ❑ 在窗体名称上单击鼠标右键，然后选择“删除”命令，可以将此窗体从当前项目内删除，如图 15-8 所示。
- ❑ 在窗体名称上单击鼠标右键，然后选择“重命名”命令，可以对此窗体进行重命名。

Form 类有如下两种窗体显示方式。

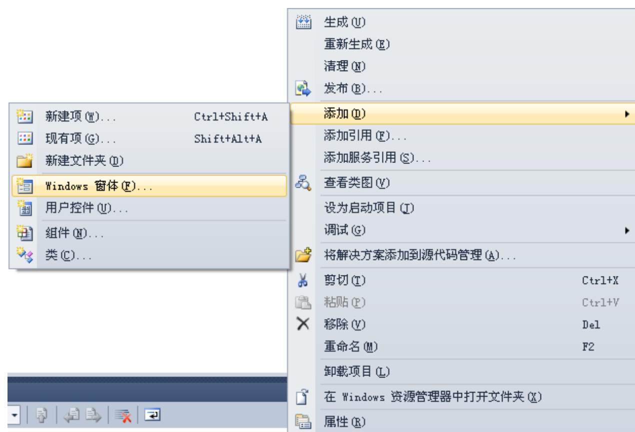


图 15-7　添加一个窗体



图 15-8　删除一个窗体

1．ShowDialog 方式

在 ShowDialog 方式下，以模式方式显示窗体，此时窗体将以独占的方式与系统和用户进行交互，用户的所有操作都被它控制。同程序内的其他窗体将被屏蔽，将不能被操作。例如下面的代码定义了一个模式窗体 mm：

```
public Form1(){
        InitializeComponent();
```

```
                FirstForm mm = new FirstForm();
                mm.ShowDialog();
            }
```

2．Show 方式

在 Show 方式下，以无模式方式显示窗体，此类型窗体可以和项目中的其他窗体进行交互，即可以同时对其他窗体进行交互。例如下面的代码定义了一个无模式窗体 Form2：

```
        public Form2(){
                InitializeComponent();
                FirstForm mm = new FirstForm();
                mm.Show();
            }
```

15.2.2 窗体设置

创建窗体后，可以对窗体的属性进行设置处理，以实现指定的处理功能。在下面的内容中将简要介绍窗体内的操作处理的基本知识，并通过一个具体的实例来说明各功能的具体实现流程。

1．消息框处理

消息框是窗体的重要组成元素之一，通过消息框可以弹出显示指定的信息。在 C#内通过 MessageBox 类的静态方法 Show（），可以显示一个包含文本、标题、按钮、图标和帮助等选项的消息框。消息框是模式对话框，所以调用 MessageBox.Show（）方法会阻止对程序其他部分的操作处理。

例如通过下面的代码，实现了在消息框内显示指定信息的效果。

```
        public Form1(){
                InitializeComponent();
                //设置消息框
                MessageBox.Show (“欢迎大家来到C#的学习世界!!! ”);
            }
```

在 C#消息框内可以指定显示按钮的内容，用户单击不同按钮可以调用不同的处理程序。MessageBoxButtons 枚举定义了消息框上显示按钮的类型，具体说明如表 15−1 所示。

表 15−1　MessageBoxButtons 枚举设置信息

枚　举　值	说　　明
AbortRetryIgnore	设置消息框包含“终止”、“重试”和“忽略”按钮
OK	设置消息框包含“确定”按钮
OKCancel	设置消息框包含“确定”和“取消”按钮
RetryCancel	设置消息框包含“重试”和“取消”按钮
YesNo	设置消息框包含“是”和“否”按钮
YesNoCancel	设置消息框包含“是”、“否”和“取消”按钮

DialogResult 枚举定义了对话框的返回值，具体说明如表 15−2 所示。

表 15-2　DialogResult 枚举设置信息

枚 举 值	说　明
Abort	返回 Abort
Cancel	返回 Cancel
Ignore	返回 Ignore
No	返回 No
None	返回 None
OK	返回 OK
Retry	返回 Retry
Yes	返回 Yes

MessageBoxIcon 枚举定义了消息框上显示的图标，具体说明如表 15-3 所示。

表 15-3　MessageBoxIcon 枚举设置信息

枚 举 值	说　明
Asterisk	显示圆圈和叹号组成的符号
Error	显示红色背景、圆圈和白色 X 组成的符号
Exclamation	显示黄色背景的三角形和叹号组成的符号
Hand	显示红色背景、圆圈和白色 X 组成的符号
Information	显示圆圈和小写 i 组成的符号
None	不包含任何符号
Question	显示圆圈和问号组成的符号
Stop	显示红色背景、圆圈和白色 X 组成的符号
Warning	显示黄色背景三角形和叹号组成的符号

2. 窗体定制处理

Form 类可以创建标准窗口、工具窗口、浮动窗口等效果。也可以使用 Form 类创建模式窗口或多文档界面窗体。设置 Form 类内的属性可以确定所创建窗口或对话框的外观样式。例如通过 Text 属性可以设置标题文字，通过 Size 可以设置窗体的大小。

窗体的边框可以通过 FormBorderStyle 属性来控制，并可以控制调整窗体大小的行为。FormBorderStyle 枚举定义了窗体的边框样式，具体说明如表 15-4 所示。

表 15-4　FormBorderStyle 枚举设置信息

枚 举 值	说　明
Fixe3D	三维边框样式显示
FixeDialog	固定对话框样式边框显示
FixeSingle	固定单行边框样式显示
FixeToolWindow	不能调整大小的工具窗口边框显示
None	没有边框
Sizeble	可调整大小的边框样式
SizebleToolWindow	可调整大小的工具窗口边框显示

当使用 FormBorderStyle 时，会在窗体的标题栏左边显示一个图标。窗体的图标相当于是窗体的图形符号，使窗体更加容易被识别。图标是一种资源图像，可以通过 Visual Studio 2010 来创建和绘制图标文件。具体流程如下。

1）打开 Visual Studio.NET 2010，然后依次选择“文件”｜“新建”｜“文件”命令，弹出“模板”对话框，如图 15-9 所示。

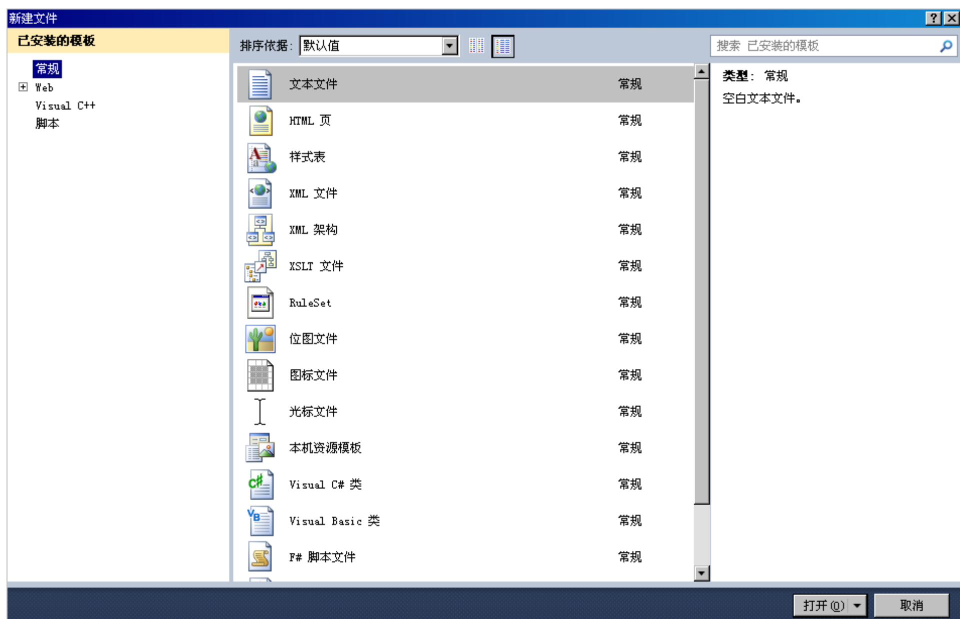


图 15-9 “模板”对话框

2）在“模板”中选择“图标文件”选项，单击“确定”按钮后将在 Visual Studio 2010 窗口内创建一个默认的图标文件，具体如图 15-10 所示。

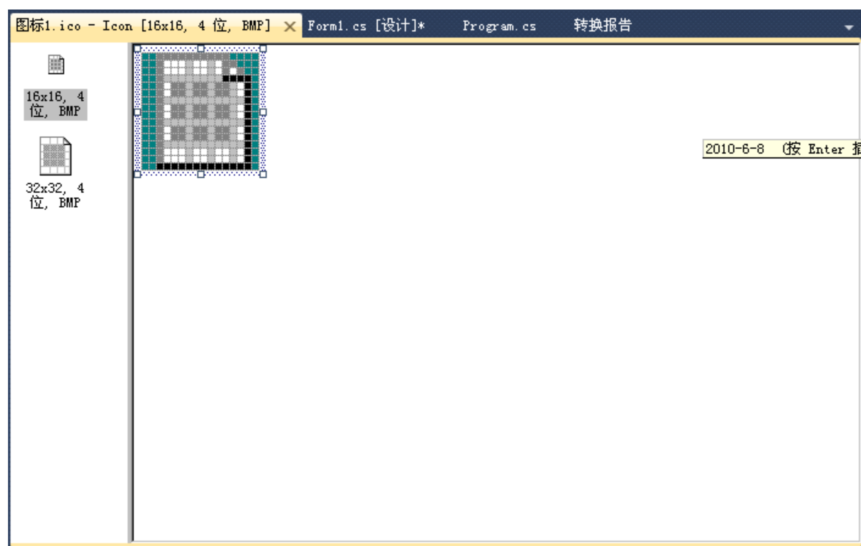


图 15-10 图标文件界面

这样通过窗体属性中的“Icon”属性，即可为当前窗体选择指定创建的图标。

3．事件处理

Windows 是事件驱动型的操作系统，对 Form 的任何操作都是基于事件驱动来实现的。在 Windows 窗体应用程序内，可以通过事件来激活对应的处理程序。一系列的可重写程序、窗体和控件可以处理这些消息。当上述方法接收到鼠标或键盘的消息后，会引发相应的事件，这样可以通过事件来获取关于鼠标或键盘的输入信息。

Form 类提供了大量的事件可以响应对窗体的各种操作。例如，窗体启动时将引发 Load 事件，窗体被激活时将引发 Activated 事件，窗体关闭时会引发 FormClosing 事件，鼠标移动时会引发 MouseMove 事件，按下键盘按键时会引发 KeyDown 事件。

如果需要在出现某事件时执行特定的操作，则需要编写特定的事件处理程序，将其包装在一个委托内，并将它和事件联系起来。例如，在窗体 Form 的构造函数内编写如下代码：

```
this.mm+=new System.EventHandler(this.nn);
```

通过上述代码就可以将事件 mm 和函数 nn()连接起来了。当出现事件 mm 时，会立即执行函数 nn()定义的内容。

实例 69：创建和设置窗体

下面将通过一个具体实例来说明创建和设置窗体的过程。本实例保存在“光盘：\daima\15”文件夹内，项目名为 FormSample，功能是在项目内创建两个窗体，并调用各属性实现对窗体的设置。本实例的具体实现流程如下。

1）打开 Visual Studio 2010，然后依次选择“文件”｜“新建”｜“项目”命令，弹出“新建项目”对话框。

2）在“模板”选项中选择“Windows 应用程序”，在名称中输入项目名“Forms”，具体如图 15-11 所示。

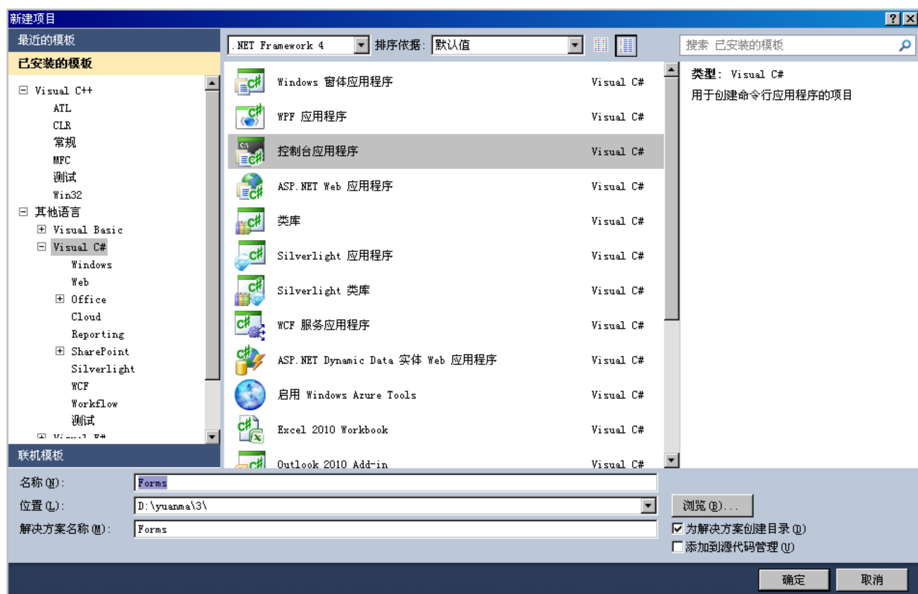


图 15-11　新建项目

3）在 Visual Studio 2010 窗体界面分别设置窗体 Form1 和 Form2 的属性，并设置相应的事件处理程序，如图 15-12 所示。

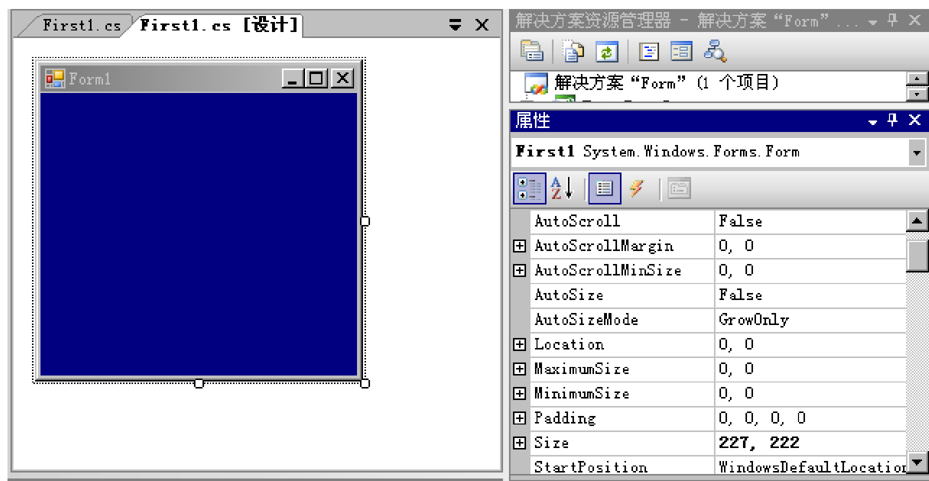


图 15-12　设置窗体 Form1

4）设计完毕后将项目保存。

下面将分别介绍各项目文件的具体实现流程。

（1）项目入口文件 Form.cs

项目执行后将首先执行入口文件 Form.cs，其功能是使用 STAThread 特性设置应用程序为单线程单元，并调用 System.Windows.Forms.Application 类的 Run()方法来启动程序。在本实例中，通过 Application.Run 设置了 Form2 类的实例作为参数，即使用 Form2 窗体对象来管理当前应用程序的信息。

文件 Form.cs 的具体实现代码如下：

```
namespace Forms
{
    static class Program
    {
        /// <summary>
        /// 应用程序的主入口点。
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new Form2());
        }
    }
}
```

（2）窗体文件 Form1.cs

窗体文件 Form1.cs 的功能是，设置窗体 Form1 的外在显示样式，并分别创建四个窗体实例，并分别设置不同的边框显示样式和事件处理程序。上述功能的具体运行流程如下：

1）设置窗体的外在显示样：通过 Location 来定位按钮 mm 的位置，通过 Text 设置 mm 的显示文本，通过 Opacity 设置窗体的透明度为 1。

2）创建了四个窗体实例：分别创建窗体实例 firstForm、secondForm、thirdForm 和 fourthForm，并分别设置各窗体实例的标题内容、边框样式和显示颜色。

3）设置事件处理程序：定义 OnmmCliecked（）方法，设置当用户单击 mm 对象时弹出信息对话框。

外在样式的设置代码如下：

```
this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 12F);
          this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
          this.BackColor = System.Drawing.SystemColors.Desktop;
          this.ClientSize = new System.Drawing.Size(219, 136);
          this.Controls.Add(this.button1);
          this.Icon=((System.Drawing.Icon)(resources.GetObject("$this.
          Icon")));
          this.Margin = new System.Windows.Forms.Padding(2);
          this.Name = "Form2";
          this.Text = "Form2";
          this.Load += new System.EventHandler(this.MainForm_Load);
          this.ResumeLayout(false);
```

对应功能的具体实现代码如下：

```
namespace Forms
{
   public partial class Form2 : Form
   {
      public Form2()
      {
         InitializeComponent();
         Button mm= new Button();
         mm.Location = new Point(24, 24);
         mm.Size = new Size(96, 24);
         mm.Text = "你好呀!";
         this.Controls.Add(mm);
         mm.Click += new EventHandler(OnmmCliecked);
         this.Opacity = 1;
// 创建 FirstForm 实例
First1 firstForm = new First1();
// 设置边框
firstForm.FormBorderStyle = FormBorderStyle.Fixed3D;
```

```
        // 以无模式方式显示 FirstForm 对象
        firstForm.Show();
        // 创建一个 Form 实例
        Form secondForm = new Form();
        // 设置 Text
        secondForm.Text = "第二个 Form";
        // 设置边框
        secondForm.FormBorderStyle = FormBorderStyle.FixedToolWindow;
        // 以无模式方式显示
        secondForm.Show();
        // 创建一个 Form 实例
        Form thirdForm = new Form();
        // 设置 Text
        thirdForm.Text = "第三个 Form";
        // 设置边框
        thirdForm.FormBorderStyle = FormBorderStyle.FixedDialog;
        // 取消最大化按钮
        thirdForm.MaximizeBox = false;
        // 以无模式方式显示
        thirdForm.Show();
        // 创建一个 Form 实例
        Form fourthForm = new Form();
        // 设置 Text
        fourthForm.Text = "第四个 Form";
        // 设置边框
        fourthForm.FormBorderStyle = FormBorderStyle.FixedDialog;
        // 设置前景色
        fourthForm.BackColor = Color.LightGray;
        // 以无模式方式显示
        fourthForm.Show();
        }
              private void OnmmCliecked(Object sender, EventArgs e)
              {
                 MessageBox.Show("按了 mm 按钮");
              }
              private void button1_Click(object sender, EventArgs e)
              {
              }
              private void MainForm_Load(object sender, EventArgs e)
              {
              }
           }
        }
```

（3）窗体文件 Form2.cs

窗体文件 Form2.cs 的功能是，设置窗体 Form2 的外在显示样式，并分别设置不同的事

件处理程序。上述功能的具体运行流程如下。

1）设置方法 DoFormClosing（），当关闭窗体时弹出指定的确认消息框。

2）定义虚方法 OnMouseDown（）来响应对鼠标的行为。

3）定义虚方法 OnKeyDown（）来响应对键盘按键的行为。

外在样式的设置代码如下：

```
private void InitializeComponent()
        {
            this.SuspendLayout();
            //
            // FirstForm
            //
            this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 12F);
            this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
            this.BackColor = System.Drawing.SystemColors.ActiveCaption;
            this.CausesValidation = false;
            this.ClientSize = new System.Drawing.Size(219, 195);
            this.Margin = new System.Windows.Forms.Padding(2);
            this.Name = "FirstForm";
            this.Text = "Form1";
            this.FormClosing += new System.Windows.Forms. FormClosing
            EventHandler(this.DoFormClosing);
            this.Load += new System.EventHandler(this.FirstForm_Load);
            this.ResumeLayout(false);
        }
```

对应处理功能的具体实现代码如下：

```
namespace Forms{
    public partial class First1 : Form
    {
        public First1()
        {
            InitializeComponent();
        }
        private void DoFormClosing(object sender,FormClosingEventArgs e)
        {
           DialogResult nn = MessageBox.Show("确认关闭窗体吗!!! ? ", "Forms",
           MessageBoxButtons.YesNo, MessageBoxIcon.Question);
            if (nn == DialogResult.No)
            {
                e.Cancel = true;
            }
        }
        protected override void OnMouseDown(MouseEventArgs e)
        {
```

```
                MessageBox.Show("单击了鼠标键: "+e.Button);
            }
            protected override void OnKeyDown(KeyEventArgs e)
            {
                MessageBox.Show("按了键盘键: " + e.KeyCode);
            }
            private void FirstForm_Load(object sender, EventArgs e)
            {
            }
        }
    }
```

上述实例代码执行后，将会弹出五个窗体框，如图 15-13 所示。

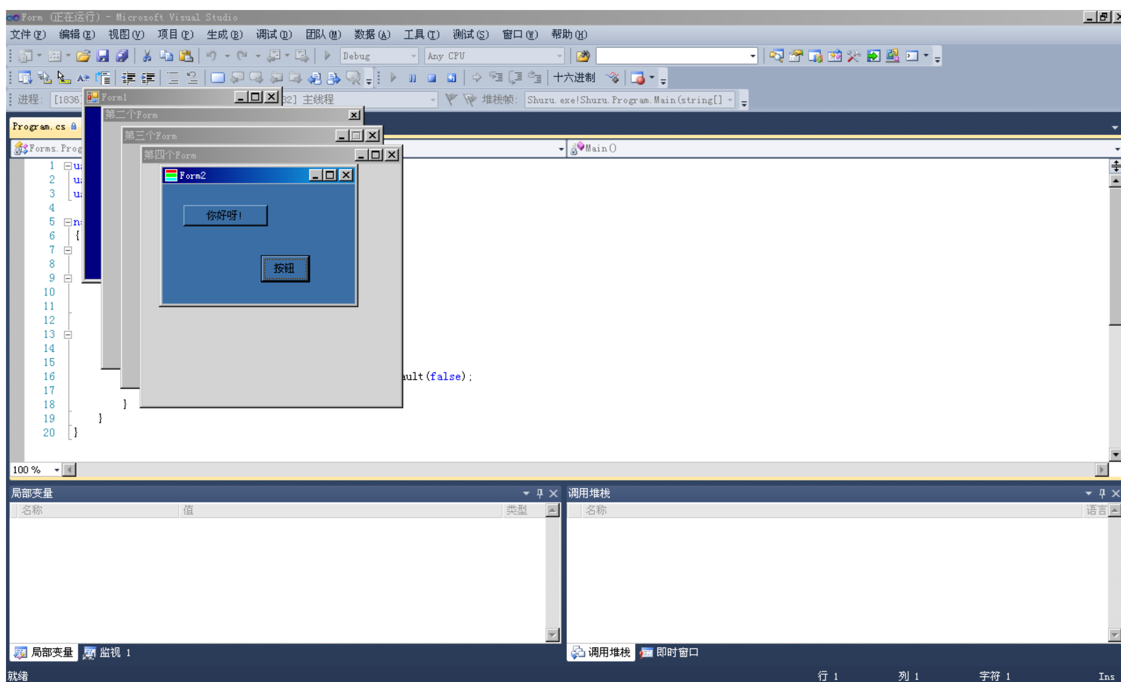


图 15-13　弹出的 5 个窗体

当在窗体 Form2 上单击按钮“你好呀！”后，会激活事件显示对应的信息框，如图 15-14 所示。

单击窗体 2、窗体 3 和窗体 4 上的关闭图标“ ✕ ”后，会将对应的窗体关闭，如图 15-15 所示。

鼠标单击窗体 Form1 后会激活事件显示对应的鼠标键值，如图 15-16 所示；按下键盘按键后会显示对应的键值，如图 15-17 所示。



图 15-14　信息框提示

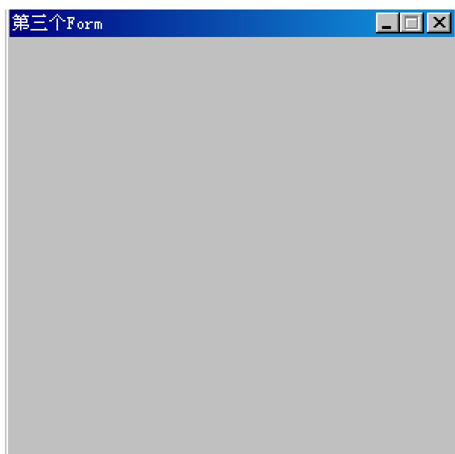


图 15-15　窗体 2、3、4 可以关闭



图 15-16　相应鼠标值

当单击窗体 Form1 上的关闭图标“×”后，会弹出确认关闭对话框，如图 15-18 所示。



图 15-17　相应按键值



图 15-18　确认关闭对话框

单击图 15-18 内的“是”按钮后将会关闭窗体 Form1，当单击窗体 Form1 上的关闭图标“×”后，将把所有的窗体关闭。

15.3　使用控件和组件

使用 System.Windows.Forms 内定义的控件，可以迅速开发出项目中所需要的功能。这样可以大大节约系统的开发时间，并使应用程序和 Windows 效果保持一致。在本节的内容中，将对 C#窗体编程常用的控件和组件知识进行简要介绍。

15.3.1　控件的属性

通过设置控件的属性可以实现控件的对应显示效果，所有的控件都有各自的属性。大多数控件的基类都是 System.Windows.Forms.Control，其他的控件属性都继承了它的属性或重写了它的属性。Control 类的常用属性信息如表 15-5 所示。

表 15-5　Control 类常用属性信息

属　性　值	说　　明
Anchor	指定控件大小变化时控件如何响应
BackColor	控件的背景颜色
Bottom	设置控件底部和窗体顶部的距离
Dock	使控件在窗口边界
Enabled	设置用户是否接受用户的输入
ForeColor	控件的前景色
Height	控件底部到顶部的距离
Left	控件左边界到窗体左边界的距离
Name	控件的名称
Parent	控件的父控件
Right	控件右边界到窗体左边界的距离
TabIndex	控件容器中的标签顺序
TabStop	设置是否可以用〈Tab〉键访问
Tag	在控件中存储控件的信息
Top	控件顶部和窗体顶部的距离
Visible	控件的可见性
Width	控件的宽度

其中属性 Anchor 和 Dock 特别有用，如果修改窗体的大小比较麻烦，而为了使窗体看上去比较美观，则可以使用属性 Anchor 和 Dock 来实现，避免了传统的编写多行代码的解决方式。属性 Anchor 指定了控件大小变化时控件如何响应。属性 Dock 使控件始终在窗口边界显示，即使用户重新设置了它的大小。



如果读者在以前使用过 Visual Basic，应该注意到在 2010 中 Text 属性用于设置显示的文本，而不用 Visual Basic 中的 Caption 属性。在 2010 推出以前，Text 是可以和 Caption 互换的。

15.3.2　使用 Label 控件和 LinkLabel 控件

Label 控件是最常用的窗体控件之一，能够在窗体上显示指定的文本。而控件 LinkLabel 也能够在窗体上显示指定的文本，但是显示的是链接形式的文本。Label 控件和 LinkLabel 控件有多个属性，其中绝大部分都是基于 Control 类的。

实例 70：使用 Label 控件和 LinkLabel 控件

下面将通过一个具体的实例来说明 Label 控件和 LinkLabel 控件的使用过程。本实例保存在“光盘：\daima\15”文件夹内，项目名为 Labelchuli，功能是在窗体内显示指定的文本。本实例的具体实现流程如下。

1）在 Visual Studio 2010 中新建一个名为“Labelchuli”的 Windows 应用程序，如图 15-19 所示。

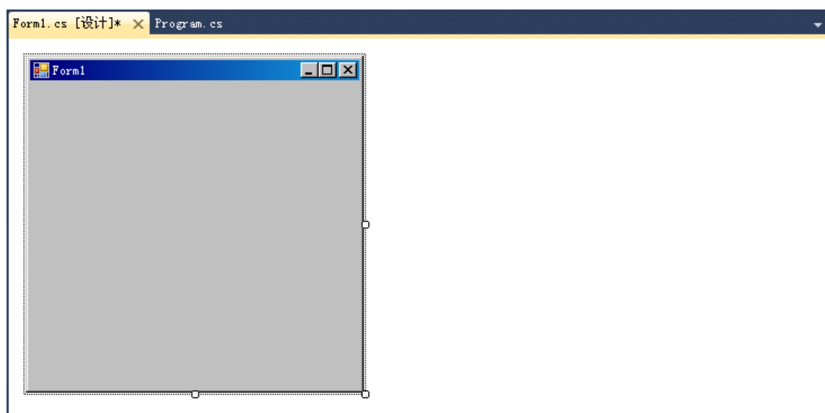


图 15-19　新建的窗体

2）从“工具箱”中拖入一个 Label 控件，设置其属性 Text 值为“这是控件的文字”，并设置字体的显示样式，如图 15-20 所示。

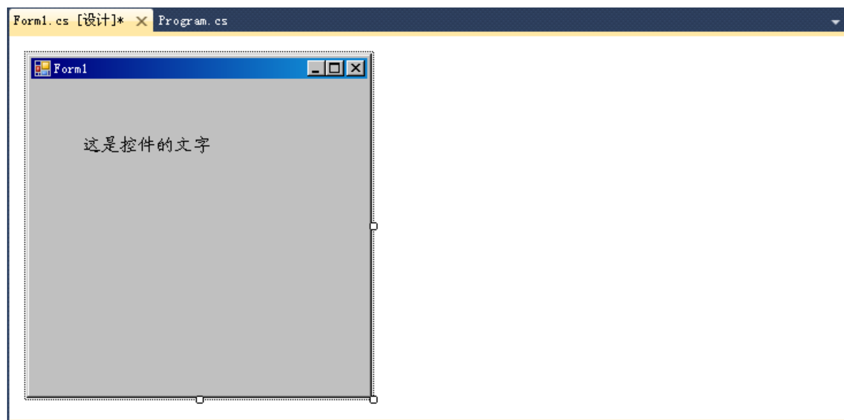


图 15-20　设置属性

3）按照上述流程再次拖入一个 LinkLabel 控件，并设置它的属性，如图 15-21 所示。

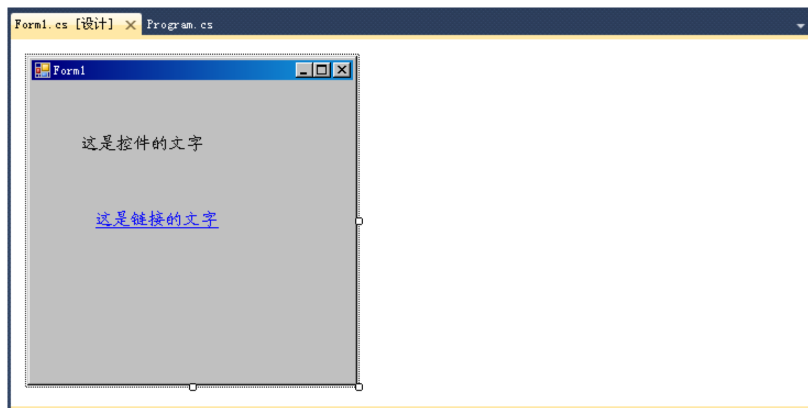


图 15-21　设置属性

4）经过上述操作后，会在窗体 Form1 内显示两段文本，分别是使用 Label 控件和 LinkLabel 控件实现的文本，如图 15-22 所示。

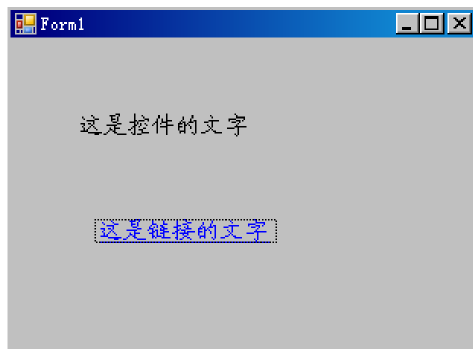


图 15-22　执行结果

下面开始分析实例的具体实现代码。

文件 Program.cs 是项目的入口程序，因为比较简单和容易理解，在此将不作介绍。文件 Form1.Designer.cs 是属性的设置文件，通过定义方法 InitializeComponent()，设置了各控件的显示属性，主要实现代码如下所示：

```
        private void InitializeComponent() {
                this.label1 = new System.Windows.Forms.Label();
                this.linkLabel1 = new System.Windows.Forms.LinkLabel();
                this.SuspendLayout();
                //
                // label1
                //
                this.label1.AutoSize = true;
                this.label1.Font = new System.Drawing.Font("楷体_GB2312",
                12F,System.Drawing.FontStyle.Regular,System.Drawing.Graphics
                Unit.Point,((byte) (134)));
                this.label1.Location = new System.Drawing.Point(42, 49);
                this.label1.Name = "label1";
                this.label1.Size = new System.Drawing.Size(120, 16);
                this.label1.TabIndex = 0;
                this.label1.Text = "这是控件的文字";
                this.label1.Click += new System.EventHandler(this.label1_Click);
                //
                // linkLabel1
                //
                this.linkLabel1.AutoSize = true;
                this.linkLabel1.Font = new System.Drawing.Font("楷体_GB2312",
                12F, System.Drawing.FontStyle.Regular, System.Drawing.Graphi
                csUnit.Point, ((byte)(134)));
                this.linkLabel1.Location = new System.Drawing.Point(55, 118);
                this.linkLabel1.Name = "linkLabel1";
```

```
            this.linkLabel1.Size = new System.Drawing.Size(120, 16);
            this.linkLabel1.TabIndex = 1;
            this.linkLabel1.TabStop = true;
            this.linkLabel1.Text = "这是链接的文字";
            //
            // Form1
            //
            this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 12F);
            this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
            this.ClientSize = new System.Drawing.Size(292, 273);
            this.Controls.Add(this.linkLabel1);
            this.Controls.Add(this.label1);
            this.Name = "Form1";
            this.Text = "Form1";
            this.ResumeLayout(false);
            this.PerformLayout();
        }
```

15.3.3 使用 TextBox 控件

控件 TextBox 是文本框显示控件，能够获取用户在窗体内输入的文本或显示的文本。TextBox 控件可以用于文本编辑处理，但只能是只读控件。文本框可以显示多行，并且可以对文本进行换行及添加基本的设置属性。TextBox 控件的常用属性信息如表 15-6 所示。

表 15-6 TextBox 常用属性信息

属 性 值	说 明
CausesValidation	当为 True 时会引发两个事件
CharacterCasing	设置是否会改变文本的大小写
MaxLength	文本的最大长度值
Multiline	设置是否是多行控件
PasswordChar	设置使用密码输入
ReadOnly	设置是否为只读
ScrollBars	多行文本是否有自动滚条
SelectedText	文本中选择的字符数
SelectionLength	文本框中被选中的文本开头
SelectionStart	设置当多行文本内的某行文本超过控件宽度时是否自动换行

其中属性 CharacterCasing 有如下三个取值。

- Lower：将文本都转换为小写。
- Upper：将文本都转换为大写。
- Normal：不对文本进行转换处理。

TextBox 控件的常用事件有如下三类。

1．焦点事件

焦点事件即 Enter、Leave、Validating 和 Validated 事件，它们将按照编写的顺序逐一被引发，当控件的焦点被改变时就会引发。但是 Validating 和 Validated 仅在接受焦点后，并 CausesValidation 值为 True 时才会被引发。

2．键事件

键事件即 KeyDown、KeyPress 和 KeyUp 事件，它们能够改变和监视输入控件中的内容。

3．都引发事件

都引发事件即 Change 事件，只要文本内容发生改变就会被引发，不论发生了什么改变。

实例 71：使用 TextBox 控件

下面将通过一个具体的实例来说明 TextBox 控件的使用过程。本实例保存在“光盘：\daima\15”文件夹内，项目名为 TextBoxchuli，功能是为窗体设置指定的显示文本。本实例的具体实现流程如下。

1）在 Visual Studio 2010 中新建一个名为“TextBoxchuli”的 Windows 应用程序。

2）从工具箱中拖入到窗体顶部一个 TextBox 控件，设置 name 为“TextBox”，设置字体 Font 大小为“12pt”，并设置字体为粗体和斜体。

3）在“属性”的“事件”界面双击 KeyDown 事件，并对自动生成的 textBox1_KeyDown()方法进行如下修改：

```
private void textBox1_KeyDown(object sender, KeyEventArgs e)
        {
            if (e.KeyCode == Keys.Enter)
                MessageBox.Show("您输入的文字为："+textBox1.Text,"TextBox
                Sample");
        }
```

4）再次拖入一个名为 textBox2 的 textBox 控件，分别设置它的属性。然后双击 TextChanged 事件，并对自动生成的 textBox2_TextChanged()方法进行如下修改：

```
private void textBox2_TextChanged(object sender, EventArgs e)
        {
            if (textBox2.Text.Length == textBox2.MaxLength)
                MessageBox.Show("您输入的密码为："+textBox2.Text,"TextBox
                Sample");
        }
```

5）再次拖入一个名为 textBox3 的 textBox 控件，分别设置它的属性。

```
private void textBox2_TextChanged(object sender, EventArgs e)
        {
            if (textBox2.Text.Length == textBox2.MaxLength)
                MessageBox.Show("您输入的密码为："+textBox2.Text, "TextBox
                Sample");
```

```
            }
```

6）经过上述操作完毕后，会在窗体 Form1 内显示 3 个不同属性样式的 TextBox 文本框，具体如图 15-23 所示。当激活 TextBox1 事件后会显示对应的提示框，如图 15-24 所示。

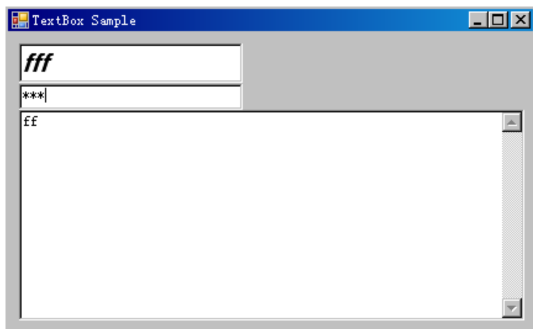


图 15-23　文本框结果



图 15-24　事件提示框

至此，整个项目设计完毕。各控件的属性设置代码在文件 Form1.Designer.cs 内，主要代码格式如下：

```
..........................................
    // textBox1
    this.textBox1.Font = new System.Drawing.Font("Microsoft Sans
    Serif",16F,((System.Drawing.FontStyle)((System.Drawing.Font
    Style.Bold|System.Drawing.FontStyle.Italic))),System.Drawing.
    GraphicsUnit.Point,((byte)(134)));
    this.textBox1.Location = new System.Drawing.Point(9, 9);
    this.textBox1.Margin = new System.Windows.Forms.Padding(2);
    this.textBox1.Name = "textBox1";
    this.textBox1.Size = new System.Drawing.Size(181, 32);
    this.textBox1.TabIndex = 0;
    this.textBox1.TextChanged += new System.EventHandler(this.
    textBox1_TextChanged);
    this.textBox1.KeyDown += new System.Windows.Forms. KeyEvent
    Handler(this.textBox1_KeyDown);
.......................................
```

事件处理代码在文件 Form1.cs 内，文件 Program.cs 是项目的入口程序。


多学一招

在应用中可以实现 TextBox 控件和 TextChanged 事件的共享。并且可以对输入的字符进行验证，例如如果只允许输入数字，则可以通过设置如下事件代码实现:

```
public TextBoxInputCheck(object sender, KeyPressEventArgs e,INPUTTYPE
type)
      {
```

```
            if(type == INPUTTYPE.INT)
            {
                string pattern = @"^[0-9]";
                Regex reg = new Regex(pattern);
                if((!reg.Match(e.KeyChar.ToString()).Success)&&(e.KeyChar.
                ToString() != " "))
                {
                    e.Handled = true;
                }
            }
            else if(type == INPUTTYPE.FLT)
            {
                string pattern = @"^[0-9]|.$";
                Regex reg = new Regex(pattern);
                if((!reg.Match(e.KeyChar.ToString()).Success)&&(e.KeyChar.
                ToString() != " "))
                {
                    e.Handled = true;
                }
                else if(e.KeyChar.ToString()=="."&&(sender as TextBox).
                Text.IndexOf('.') > 0)
                {
                    e.Handled = true;
                }
            }
        }
```

15.3.4 使用 Button 控件

控件 Button 是按钮控件，能够在窗体上建立一个按钮，并且允许用户通过单击来完成指定的操作。每当用户单击按钮后，就会调用 Click 事件处理方法。

单击 Button 控件后还会生成其他事件，例如 MouseEnter、MouseDown 等。如果要为这些事件设置相关的事件处理程序，必须确保它们之间的操作不会冲突。另外，Button 控件不支持双击事件，当用户双击后，会连续处理两次单击事件。Button 控件的常用属性如表 15-7 所示。

表 15-7 Button 常用属性信息

属 性 值	说 明
FlatStyle	设置按钮的显示样式，例如可以设置为平面或 3D 外观
Enabled	设置按钮是否灰色显示
Image	设置按钮上的图像
ImageAlign	设置按钮上图像的位置

其中，在按钮上显示的文本由属性 Text 确定，当文本超出按钮宽度时会自动换行显

示。但是如果当前控件无法包含文本的总体高度，则会剪裁显示。允许用户通过同时按下〈Alt〉键和访问键开启控件，如果要定义访问键则需要在访问键字符前加“&”字符。

实例 72：使用 Button 控件

下面将通过一个具体的实例来说明 Button 控件的使用过程。本实例保存在“光盘：\daima\15”文件夹内，项目名为 ButtonSample，功能是为窗体内的按钮设置指定的样式。本实例的具体实现流程如下。

1）在 Visual Studio 2010 中新建一个名为“ButtonSample”的 Windows 应用程序。

2）从工具箱中拖入到窗体顶部一个 Button 控件，设置 name 为“button1”，设置 Text 属性为“按钮 1”，设置字体 Font 大小为 16，并设置字体为粗体和斜体。

3）在窗体上调整按钮的大小和位置，让设置的文本能够完整的显示出来。

4）在“属性”的“事件”界面双击 Click 事件，并对在 Form1.cs 中自动生成的 textBox1_KeyDown（）方法进行如下修改：

```
        private void button1_Click(object sender, EventArgs e)
               {
                   MessageBox.Show("你单击了<按钮>", "Button Sample");
               }
```

5）从工具箱中拖入到窗体中间一个 Button 控件，设置 name 为“button2”，设置 Text 属性为空值，设置属性 Image 为“ButtonSample.Properties.Resources.water”。

6）在窗体上调整按钮的大小和位置，使之和“button1”的宽度一致。

7）在“属性”的“事件”界面双击 Click 事件，并对在 Form1.cs 中增加如下代码：

```
        private void button2_Click(object sender, EventArgs e)
               {
                   this.Opacity -= 0.2;
                   if (this.Opacity <= 0.1)
                       this.Opacity = 1;
               }
```

8）从工具箱中再次拖入到窗体底部一个 Button 控件，设置 name 为“button3”，设置 Text 属性为“退出”，设置 Image 属性为“ButtonSample.Properties.Resources.water”，设置 BackColor 属性为“ActiveCaptionText”。

9）在窗体上调整按钮的大小和位置，使其在“button2”下显示。

10）在设计界面选中窗体 Form1，在“属性”中设置 CancelButton 属性值为“button3”。

11）在“属性”的“事件”界面双击 Click 事件，并在 Form1.cs 中增加如下代码：

```
        private void button3_Click(object sender, EventArgs e)
               {
                   this.Close();
               }
```

12）从工具箱中拖入到窗体底部一个 TextBox 控件，设置 name 为“textBox1”，设置 Text 属性值为“测试接受/取消按钮，单击此文本框，再按〈Enter〉键或〈Esc〉键”，设置

属性 ReadOnly 值为 True。

13）经过上述操作处理后，整个项目设计完毕。最终的设计界面如图 15-25 所示。



图 15-25 项目设计界面结构

当上述项目运行后，会按照指定的样式显示窗体的各个元素，如图 15-26 所示。当单击“按钮 1”时会弹出指定的对话框，如图 15-27 所示。当单击中间按钮“按钮 2”时会使窗体的透明度发生变化，如图 15-28 所示。当单击底部的按钮“退出”时会退出当前的窗体。



图 15-26 窗体界面效果



图 15-27 弹出对话框



图 15-28 窗体透明度变化

至此，整个项目设计完毕。各控件的属性设置代码在文件 Form1.Designer.cs 内，主要代码格式如下：

```
……………………………………
// button1
          this.button1.BackColor = System.Drawing.SystemColors.Desktop;
          this.button1.Font=new System.Drawing.Font("Microsoft Sans
          Serif",16F,((System.Drawing.FontStyle)((System.Drawing.Font
          Style.Bold|System.Drawing.FontStyle.Italic))),System.Drawing.
          GraphicsUnit.Point,((byte)(134)));
          this.button1.Location = new System.Drawing.Point(70, 11);
          this.button1.Margin = new System.Windows.Forms.Padding(2, 2, 2, 2);
          this.button1.Name = "button1";
          this.button1.Size = new System.Drawing.Size(306, 34);
          this.button1.TabIndex = 0;
          this.button1.Text = "按钮 1";
          this.button1.UseVisualStyleBackColor = false;
          this.button1.Click += new System.EventHandler(this.button1_Click);
……………………………………
```

事件处理代码在文件 Form1.cs 内，而文件 Program.cs 是项目的入口程序。

15.3.5　使用 CheckBox、RadioButton 和 GroupBox 控件

控件 CheckBox 是复选框控件，能够在窗体上提供复选框按钮；控件 RadioButton 是单选按钮控件，能够在窗体上提供单选按钮；控件 GroupBox 是分组框控件，能够在窗体上提供分组框按钮。其中 CheckBox 控件和 RadioButton 控件常被用于信息收集项目中，例如用户注册平台。分组框控件也非常有用，它可以使成组的多个控件在窗体器上一起移动。

1．CheckBox 控件属性和事件

CheckBox 控件的常用属性如下。

1）CheckState：设置复选框的状态，复选框具有如下三种状态：

- Checked：表示此复选框默认被选中。
- Indeterminate：表示此复选框是灰色无效的。
- Unchecked：表示此复选框没有被选中。

2）ThreeState：当此属性值为 false 时，用户将不能把 CheckState 设置为 Indeterminate，但是可以在代码中把 CheckState 设置为 Indeterminate。

CheckBox 控件的常用事件如下。

1）CheckChanged：设置被选中时将会引发事件。

2）ChangeStateChanged：设置每当 ChangeState 属性被改变时就会引发事件。

2．RadioButton 控件属性和事件

RadioButton 控件的常用属性如下。

- Appearance：设置控件为圆形选中标签或显示为标准按钮，当被选中时会显示对应的按下状态。
- AutoCheck：为 True 时，被单击后会显示选中标记；为 False 时，必须在事件处理程

序中手工检查。

- CheckAlgin：设置对齐方式。
- Checked：设置被选中的状态。

RadioButton 控件的常用事件如下。

- CheckChanged：设置被选中时是否将会引发事件。
- Click：设置是否每当被单击时就会引发事件，无论多少次都会引发。

因为 GroupBox 控件的功能是将上述的 CheckBox 控件和 RadioButton 控件组合到一起，所以只需将被操作控件拖到 GroupBox 里面即可。

实例 73：使用 CheckBox、RadioButton 和 GroupBox 控件

下面将通过一个具体的实例来说明 CheckBox、RadioButton 和 GroupBox 控件的综合使用过程。本实例保存在“光盘：\daima\15”文件夹内，项目名为 fuxuan，功能是在窗体内设置多个按钮选项。本实例的具体实现流程如下。

1）在 Visual Studio 2010 中新建一个名为“fuxuan”的 Windows 应用程序。

2）从工具箱中拖入到窗体顶部一个 GroupBox 控件，设置 name 为“groupBox1”，设置 Text 属性为“请输入您的基本信息”，设置字体 Font 为“宋体, 9pt”。

3）从工具箱中分别拖入到 groupBox1 内四个 CheckBox 控件，并进行如下设置。

- 第一个 CheckBox：设置属性 name 值为“checkBox1”，设置 Text 属性为“AAAAAAA”。
- 第二个 CheckBox：设置属性 name 值为“checkBox2”，设置 Text 属性为“BBBBBB”。
- 第三个 CheckBox：设置属性 name 值为“checkBox3”，设置 Text 属性为“CCCCCC”。
- 第四个 CheckBox：设置属性 name 值为“checkBox4”，设置 Text 属性为“DDDDDD”。

4）分别选择上述 4 个 CheckBox 控件，在“属性”的“事件”界面双击 CheckChanged 事件，并在 Form1.cs 中为上述控件分别添加对应的事件处理方法，具体代码如下：

```
        private void checkBox1_CheckedChanged(object sender, EventArgs e)
            {
               CheckBox cb = sender as CheckBox;
               string sel = cb.Checked ? "您选择了" : "您没有选择";
               MessageBox.Show(sel + "<" + cb.Text + ">", "Group Sample");
            }
            private void checkBox2_CheckedChanged(object sender,EventArgs e)
            {
               CheckBox cb = sender as CheckBox;
               string sel = cb.Checked ? "您选择了" : "您没有选择";
               MessageBox.Show(sel + "<" + cb.Text + ">", "Group Sample");
            }
            private void checkBox3_CheckedChanged(object sender,EventArgs e)
            {
               CheckBox cb = sender as CheckBox;
```

```
            string sel = cb.Checked ? "您选择了" : "您没有选择";
            MessageBox.Show(sel + "<" + cb.Text + ">", "Group Sample");
        }
        private void checkBox4_CheckedChanged(object sender,EventArgs e)
        {
            CheckBox cb = sender as CheckBox;
            string sel = cb.Checked ? "您选择了" : "您没有选择";
            MessageBox.Show(sel + "<" + cb.Text + ">", "Group Sample");
        }
```

5）再次从工具箱中拖入到窗体中间一个 GroupBox 控件，设置 name 为“groupBox2”，设置 Text 属性为“您的性别”，设置字体 Font 为“宋体, 9pt”。

6）从工具箱中分别拖入到 groupBox2 内两个 RadioButton 控件，并进行如下设置。

- 第一个 RadioButton：设置属性值为“radioButton1”，设置 Text 属性为“男”。
- 第二个 CheckBox：设置属性值为“radioButton12”，设置 Text 属性为“女”。

7）分别选择上述两个 RadioButton 控件，在“属性”的“事件”界面双击 CheckChanged 事件，并在 Form1.cs 中为上述控件分别添加对应的事件处理方法，具体代码如下：

```
    private void radioButton1_CheckedChanged(object sender, EventArgs e)
        {
            RadioButton  rb = sender as RadioButton;
            string sel = rb.Checked ? "您选择了" : "您没有选择";
            MessageBox.Show(sel + "<" + rb.Text + ">", "Group Sample");
        }
    private void radioButton2_CheckedChanged(object sender, EventArgs e)
        {
            RadioButton rb = sender as RadioButton;
            string sel = rb.Checked ? "您选择了" : "您没有选择";
            MessageBox.Show(sel + "<" + rb.Text + ">", "Group Sample");
        }
```

8）从工具箱中分别拖入到窗体底部 1 个 Button 控件，设置属性 name 值为“button1”，设置 Text 属性为“显示收集到的信息”。

9）选中 button1，在“属性”的“事件”界面双击 Click 事件，并在 Form1.cs 中为上述控件分别添加对应的事件处理方法，具体代码如下：

```
    private void button1_Click(object sender, EventArgs e)
        {
            string sel1=this.checkBox1.Checked?(this.checkBox1.Text+"\n"):"";
            string sel2=this.checkBox2.Checked?(this.checkBox2.Text+"\n"):"";
            string sel3=this.checkBox3.Checked?(this.checkBox3.Text+"\n"):"";
            string sel4=this.checkBox4.Checked?(this.checkBox4.Text+"\n"): "";
            string sel5 = this.groupBox2.Text + "为";
            string sel6 = this.radioButton1.Checked ? this.radioButton1.
```

```
                Text : this.radioButton2.Text;
                MessageBox.Show(sel1 + sel2 + sel3 + sel4 + sel5 + sel6,
                "Group Sample");
            }
```

10）经过上述操作处理后，整个项目设计完毕。完成后的具体设计界面如图 15-29 所示。



图 15-29　项目设计界面结构

当上述项目运行后，会按照指定的样式显示窗体的各个元素，如图 15-30 所示。当单击选择某选项时会弹出对应的提示对话框，如图 15-31 所示。当选择完毕并单击按钮“显示收集信息”后，会弹出提示对话框，并在对话框内显示已选择的信息，如图 15-32 所示。



图 15-30　窗体界面效果



图 15-31　弹出对话框



图 15-32 显示已选择的信息

至此，整个项目设计完毕。各控件的属性设置代码在文件 Form1.Designer.cs 内，主要代码格式如下：

```
……………………………………
this.checkBox1.AutoSize = true;
          this.checkBox1.Location = new System.Drawing.Point(21, 20);
          this.checkBox1.Margin = new System.Windows.Forms.Padding(2);
          this.checkBox1.Name = "checkBox1";
          this.checkBox1.Size = new System.Drawing.Size(66, 16);
          this.checkBox1.TabIndex = 0;
          this.checkBox1.Text = "AAAAAAA";
          this.checkBox1.UseVisualStyleBackColor = true;
          this.checkBox1.CheckedChanged+=new System.EventHandler (this.
          checkBox1_CheckedChanged);
……………………………………
this.groupBox2.Controls.Add(this.radioButton2);
          this.groupBox2.Controls.Add(this.radioButton1);
          this.groupBox2.Location = new System.Drawing.Point(9, 128);
          this.groupBox2.Margin = new System.Windows.Forms.Padding(2);
          this.groupBox2.Name = "groupBox2";
          this.groupBox2.Padding = new System.Windows.Forms.Padding(2);
          this.groupBox2.Size = new System.Drawing.Size(201, 68);
          this.groupBox2.TabIndex = 1;
          this.groupBox2.TabStop = false;
          this.groupBox2.Text = "您的性别";
……………………………………
```

事件处理代码在文件 Form1.cs 内，而文件 Program.cs 是项目的入口程序。

15.3.6 使用 ListBox 和 ComboBox 控件

ListBox 控件和 ComboBox 控件是重要的列表控件之一。ListBox 控件能够在窗体内显示列表框，用户可以选择列表内的一项或多个选项。如果列表项数超过了可显示的数目，会自动增加显示列表滚条。ComboBox 控件的功能是，显示与一个 ListBox 组合的文本框编辑字段，该字段存在一个隐藏的下拉列表。

ListBox 控件的常用属性信息如表 15-8 所示。

表 15-8 ListBox 常用属性信息

属 性 值	说 明
SelectedIndex	如果有多个选项，则此选项是第一个
ColumnWidth	设置列的宽度
Items	包含列表框中所有的选项
MultiColumn	获取列表框中选项的列数
SelectedIndices	包含列表框中选项所有基于 0 的索引
SelectionMode	设置可以使用 List SelectionMode 枚举中的模式
Sorted	如果值为 True，则设置项目按字母顺序排序
Text	文本值

其中，ListSelectionMode 枚举具有如下四种模式。

1）None，不使用任何模式。

2）One，设置只能使用一个选项。

3）MultiSimple，可以使用多个选项。

4）MultiExtended，可以使用多个选项，并且可以使用 Ctrl、Shift 和箭头灯键进行选择。

通过属性 Sorted 可以设置是否对 ListBox 和 ComboBox 的列表选项进行排序，如果值为 True，则将强制集合中各项重新添加到控件中，并将各项插入到指定的位置。

15.3.7 使用 ProgressBar 和 BackgroudWorker

ProgressBar 控件即进度条控件，能够在水平方框内显示合适数目的矩形符号，这些符号用于指明当前某项工作的进度。例如，常见的进度条控件应用是安装进度和下载进度。而 BackgroudWorker 是一个组件，也被常用于进度操作的项目，其功能是用于异步执行需要多长时间能够完成操作。

ProgressBar 控件的常用属性如下：

- Value：表示操作过程已经完成的进度。
- Minimum：设置进度条可以显示的最小值。
- Maixmum：设置进度条可以显示的最大值。

通常使用代码来设置 Value 的值，这样可以更新当前的进度值。另外也可以设置 Step 属性来指定 Value 的递增值。

如果使用 ProgressBar 控件来表示工作进度所需要的时间，上述操作会经常导致用户操

作运行界面的挂起。为了解决上述问题，在.NET 框架中引入了 BackgroudWorker 控件。通过 BackgroudWorker 控件，可以在不同程序的主用户界面线程的另一线程上异步执行耗时的操作，这样就分担了 ProgressBar 的工作负担，从而避免了挂起。



注意

如果在窗体中使用组件，从工具箱中拖入的组件会在窗体的组件栏中显示，如图 15-33 所示。

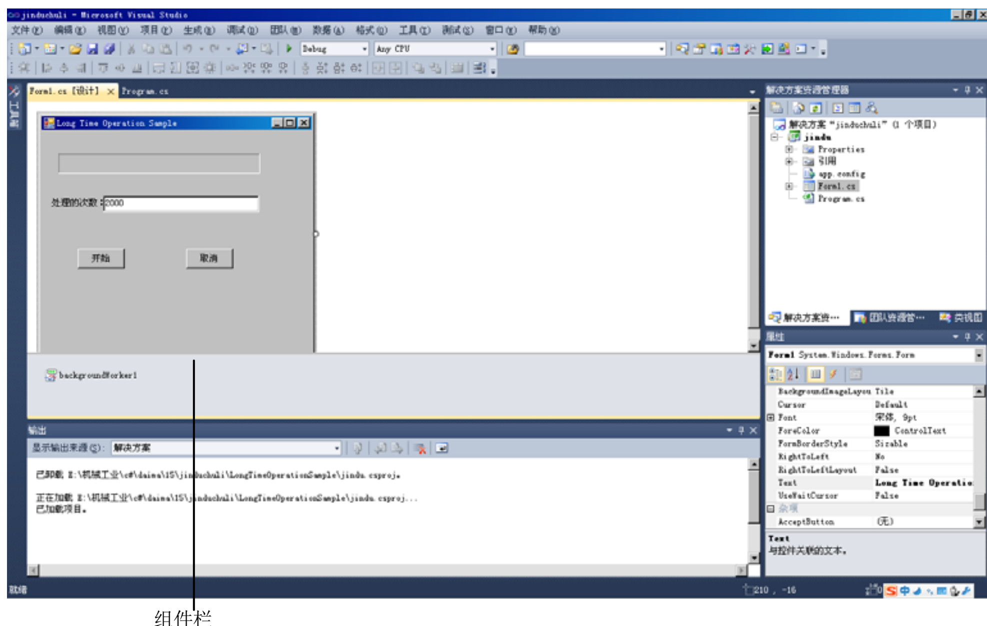


图 15-33　窗体的组件栏

实例 74：使用 ProgressBar 和 BackgroudWorker 控件

下面将通过一个具体的实例来说明，ProgressBar 和 BackgroudWorker 的综合使用过程。本实例保存在“光盘：\daima\15”文件夹内，项目名为 jinduchuli，功能是在窗体设置一个进度框，并统计对应处理所需要的时间。本实例的具体实现流程如下。

1）在 Visual Studio 2010 中新建一个名为“jinduchuli”的 Windows 应用程序。

2）从工具箱中拖入到窗体顶部一个 ProgressBar 控件，设置 name 为“progressBar1”，并调整它的大小。

3）从工具箱中拖入到 progressBar1 下方一个 Label 控件，设置 name 为“label1”，设置 Text 属性值为“处理的次数：”。

4）从工具箱中分别拖入到窗体底部两个 Button 控件，从左到右依次进行如下设置。

- 第一个 Button：设置属性 name 值为“button1”，设置 Text 属性为“开始”。
- 第二个 Button：设置属性 name 值为“button2”，设置 Text 属性为“结束”。

5）从工具箱中拖入一个 BackgroudWorker 组件，设置 name 为“backgroundWorker1”，设置 WorkerReportsProgress 属性值为“True”，设置 WorkerSupportsCancell 属性值为

"True"。

6）选中 backgroundWorker1，在"属性"的"事件"界面分别双击 DoWork、ProgressChanged 和 RunWorkerCompleted 事件，并在 Form1.cs 中为上述组件添加对应的事件处理方法，具体代码如下：

```
        private void button1_Click(object sender, EventArgs e)
        {
            button1.Enabled = false;
            button2.Enabled = true;
            long l = 1000;
            try
            {
                l = Convert.ToInt64(textBox1.Text);
            }
            catch
            {
            }
            backgroundWorker1.RunWorkerAsync(l);
        }
        private void button2_Click(object sender, EventArgs e)
        {
            // 取消异步操作
            this.backgroundWorker1.CancelAsync();
            button2.Enabled = false;
        }
        private void backgroundWorker1_DoWork(object sender, DoWorkEventArgs e)
        {
            // 获取产生本事件的 BackgroundWorker 对象
            BackgroundWorker worker = sender as BackgroundWorker;
            // 将计算结果赋给 DoWorkEventArgs 的 Result 属性
            // RunWorkerCompleted 事件的处理方法将得到该值
            e.Result = LongTimeOperation((long)e.Argument, worker, e);
        }
        private void backgroundWorker1_ProgressChanged(object sender,
        ProgressChangedEventArgs e)
        {
            this.progressBar1.Value = e.ProgressPercentage;
        }
        private void backgroundWorker1_RunWorkerCompleted(object sender,
        RunWorkerCompletedEventArgs e)
        {
            // 首先处理出现错误或者异常的情形
            if (e.Error != null)
            {
                MessageBox.Show(e.Error.Message,"Long Time Operation Sample");
            }
```

```
        else if (e.Cancelled)    // 处理被用户取消的情形
        {
            MessageBox.Show("用户取消了操作","Long Time Operation Sample");
        }
        else                    // 正确得到了结果
        {
            MessageBox.Show("最终的处理结果为：" + e.Result.ToString(),
            "Long Time Operation Sample");
        }
        button1.Enabled = true;
        button2.Enabled = false;
        this.progressBar1.Value = 0;
    }
    //进行实际工作的方法
    // 进行循环加法计算
    private long LongTimeOperation(long maxNum, BackgroundWorker
    worker, DoWorkEventArgs e)
    {
        long result = 0;
        for (long l = 0; l < maxNum; ++l)
        {
            // 如果用户取消运算，则终止操作
            if (worker.CancellationPending)
            {
                e.Cancel = true;
            }
            else
            {
                result += l;
                // 报告已完成工作的百分比
                int percentComplete =
                    (int)((float)l / (float)maxNum * 100);
                worker.ReportProgress(percentComplete);
                // 让出时间片给主线程
                Thread.Sleep(1);
            }
        }
        return result;
    }
```

7）经过上述操作处理后，整个项目设计完毕。最终的具体设计界面如图 15-34 所示。

当上述项目运行后，会按照指定的样式显示窗体的各个元素，如图 15-35 所示。当单击“开始”按钮后会显示对应的处理进度条，如图 15-36 所示。处理完毕后会弹出显示结果的对话框，如图 15-37 所示。当在处理次数框内输入另外的数据并单击“开始”按钮后，也会进行对应的进度处理。

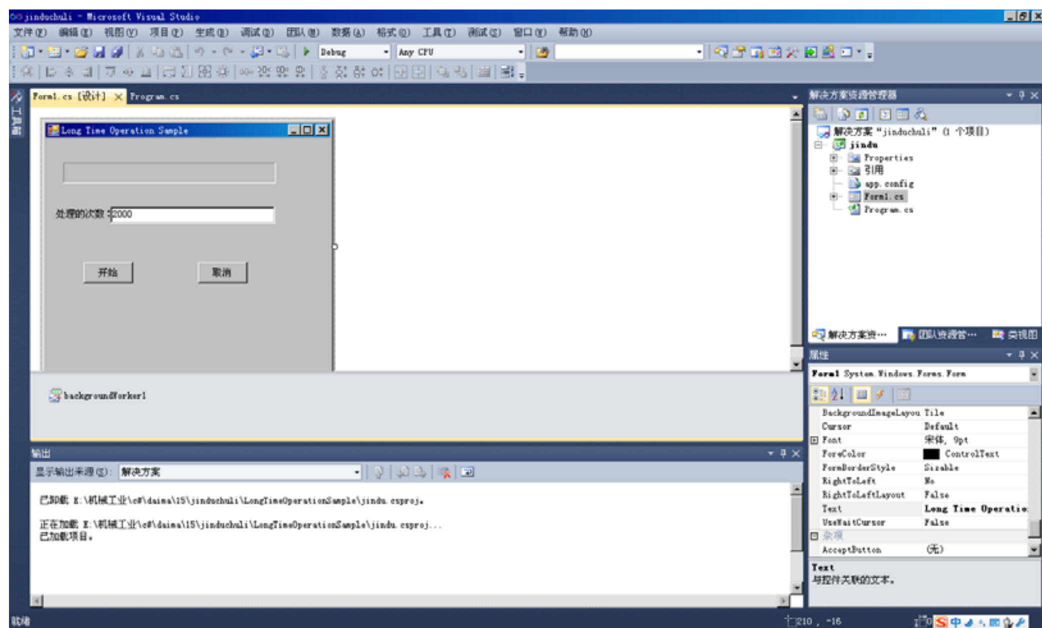


图 15-34　项目设计界面结构



图 15-35　窗体界面效果



图 15-36　处理进度条

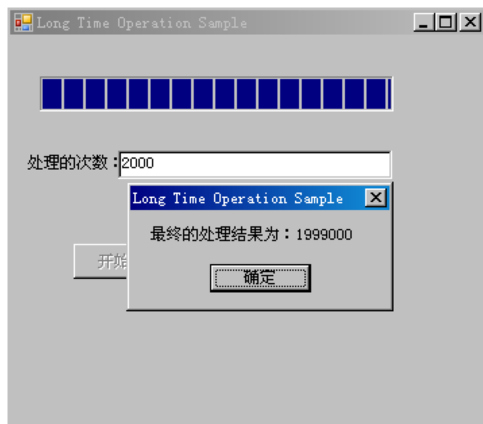


图 15-37　处理结果对话框

至此，整个项目设计完毕。各控件的属性设置代码在文件 Form1.Designer.cs 内，主要代码格式如下：

```
        ……………………………………………
            // button2
            this.button2.Location = new System.Drawing.Point(188, 154);
            this.button2.Margin = new System.Windows.Forms.Padding(2);
            this.button2.Name = "button2";
            this.button2.Size = new System.Drawing.Size(61, 26);
            this.button2.TabIndex = 2;
            this.button2.Text = "取消";
            this.button2.UseVisualStyleBackColor = true;
            this.button2.Click += new System.EventHandler(this.button2_
            Click);
            this.backgroundWorker1.WorkerReportsProgress = true;
            this.backgroundWorker1.WorkerSupportsCancellation = true;
            this.backgroundWorker1.DoWork+=new System.ComponentModel.Do
            WorkEventHandler(this.backgroundWorker1_DoWork);
            this.backgroundWorker1.RunWorkerCompleted+=new System.Compon
            entModel.RunWorkerCompletedEventHandler(this.backgroundWor
            ker1_RunWorkerCompleted);
            this.backgroundWorker1.ProgressChanged+=new System.Component
            Model.ProgressChangedEventHandler(this.backgroundWorker1_Pro
            gressChanged);
            // label1
            this.label1.AutoSize = true;
            this.label1.Location = new System.Drawing.Point(11, 88);
            this.label1.Margin = new System.Windows.Forms.Padding(2, 0,
            2, 0);
            this.label1.Name = "label1";
            this.label1.Size = new System.Drawing.Size(77, 12);
            this.label1.TabIndex = 3;
            this.label1.Text = "处理的次数：";
            // textBox1
            this.textBox1.Location = new System.Drawing.Point(80, 85);
            this.textBox1.Margin = new System.Windows.Forms.Padding(2);
            this.textBox1.Name = "textBox1";
            this.textBox1.Size = new System.Drawing.Size(203, 21);
            this.textBox1.TabIndex = 4;
            this.textBox1.Text = "2000";
    ………………………………………………………
```

事件处理代码在文件 Form1.cs 内，而文件 Program.cs 是项目的入口程序。



多学一招

在实际软件项目开发过程中，经常会使用 ProgressBar 控件实现进度条效果。如果要显

示按固定时间间隔增长的进度，则可以设置该值，然后调用方法，使 ProgressBar 控件的值按该时间间隔递增。对于计时器以及其他一些无法以整体的百分比测量进度的方案，这是非常有用的。具体来说有如下两方面应用。

（1）使进度栏按固定值递增

设置 ProgressBar 控件的 Minimum 和 Maximum 值，将控件的 Step 属性设置为整数，该整数代表进度栏的显示值递增的数量，然后调用 PerformStep（ ）方法，使显示值按 Step 属性中设置的数量进行更改。

（2）使进度栏按动态值递增

设置 ProgressBar 控件的 Minimum 和 Maximum 值，调用 Increment（ ）方法，使显示值按指定的整数进行更改。

15.3.8 使用 ListView 控件

ListView 控件是一个列表数据显示控件。通过 ListView 控件可以实现类似于 Windows 资源管理器的显示界面。ListView 控件的常用属性如下所示。

- ❑ Activation：设置用户在列表视图中激活选项的方式，主要有如下三种取值。
- ● Standard：用户为个人机器选择的值。
- ● OneClick：单击某个选项，激活它。
- ● TowClick：双击某个选项，激活它。
- ❑ Alignment：设置列表视图中选项的对齐方式，主要有如下四种方式。
- ● Default：选项被拖放后仍位于原来的位置。
- ● Left：设置选项左对齐。
- ● Top：设置选项顶部对齐。
- ● SnapToGrid：创建一个不可见网格，选项被放在网格内。
- ❑ AutoArrange：如果设置为 True，则按照 Alignment 的设置进行排序。
- ❑ CheckBoxes：如果设置为 True，则列表中每个选项会在其左边显示一个复选框。
- ❑ CheckedIndices 和 CheckedItems：分别访问索引和选项集合。
- ❑ Columns：对列表视图中的列进行处理。
- ❑ FocusedItem：包含列表视图中有焦点的选项。
- ❑ FullRowSelected：如果设置为 True，则单击某选项后此选项会突出显示。
- ❑ GridLines：如果设置为 True，则在列表视图内绘制网格线。
- ❑ HeadStyle：设置列标题的显示样式，具体说明如下。
- ● Clickable：标题显示为一个按钮。
- ● NoClickable：标题不响应鼠标单击。
- ● None：不显示列表标题。
- ❑ Items：列表选项集合。
- ❑ LabelEdit：如果设置为 True，则可以编辑第一列的内容。
- ❑ LabelWrap：如果设置为 True，则标签会自动换行。
- ❑ MultiSelect：如果设置为 True，则可以选择多个选项。
- ❑ Scrollable：如果设置为 True，则显示滚条。

- Sorting：可以对列表选项进行排序。
- SelectedIndices 和 SelectedItems：包含选中索引和选项的集合。
- SmallImageList：当属性 View 为 SmallCon 时，此属性包含了 ImageList，ImageList 包含了要使用的图像。
- TopItem：返回列表视图的顶部选项。
- View：设置选项的显示模式，有如下五种模式。
  - LargeIcon：在所有选项旁边显示一个 32×32 的图标和一个标签。
  - SmallIcon：在所有选项旁边显示一个 16×16 的图标和一个标签。
  - List：只显示一列，该列可以包含图标和标签。
  - Details：可以显示任意数量的列，并且只有第一列可以包含图标。
  - Title：显示一个 32×32 的图标和一个标签，在图标右侧显示其子项的信息。



在传统描述下，View 设置的选项显示模式有四种，即没有把 Title 方式划入到里面。这是因为 Title 模式只能在 XP 系统以上才能起作用，而在其他平台上不能起到设置的作用。

ListView 控件的常用方法如下。

- BeginUpdate（）方法：告诉列表视图停止更新，直到调用 EndUpdate 为止。
- Clear（）方法：清除列表视图，删除所有选项和列。
- EndUpdate（）方法：在调用 BeginUpdate 后调用它。
- EnsureVisible（）方法：调用后列表视图会滚动，显示指定的索引项。
- GetItemAt（）方法：返回列表视图中唯一（x,y）的选项。

ListView 控件的常用事件如下所示。

- AfterLabelEdit（）方法：编辑标签后引发这个事件。
- BeforeLabelEdit（）方法：编辑标签前引发这个事件。
- ColumnClick（）方法：单击列时引发这个事件。
- ItemActivate（）方法：激活选项时引发这个事件。

15.3.9　使用 TreeView 控件

TreeView 控件是树视图控件，功能是用于显示节点的层次结构效果，并且能够在各节点内显示对应的子节点。用户可以展开或以折叠的方式将子节点信息显示出来。

TreeView 控件主要通过如下两个属性实现树形功能。

- Nodes：包含树视图中的顶节点列表，并且包含了 TreeNode 对象的集合，每个对象都具有一个 Nodes 属性，每个属性都可以包含自己的 TreeNodeCollection。
- SelectedNode：设置当前选中的节点。

树视图中的每个对象都具有可用于定位树视图的属性，例如 FirstNode、LastNode、NextNode、PrevNode 和 Parent。

如果某个节点有子节点，则将子节点放到它的 Nodes 属性中。TreeView 控件本身具有 TopNode 属性，此属性是整个树视图的根节点。使用递归方法可以访问树视图中的每个节点，例如下面的代码使用了递归访问：

```
        private void AccessNodeRecursive(TreeNode treeNode, List<TreeNode> nodes)
        {
            // 处理每一个节点
            if (treeNode.Checked)
                nodes.Add(treeNode);
            // 对每一个子节点递归
            foreach (TreeNode tn in treeNode.Nodes)
            {
                AccessNodeRecursive(tn, nodes);
            }
        }
```

在 TreeView 控件内可以显示指定的图像，首先将 ImageList 分配给 TreeView 的 ImageList 属性，然后通过引用 Image 在 ImageList 中的位置来分配该 Image。可以使用如下两个属性实现图像的分配。

- 属性 ImageIndex：设置当树节点未被选定时所显示的 Image 的索引值。
- 属性 SelectedImageIndex：设置当树节点被选定时所显示的 Image 的索引值。

控件 TreeView 可以使用分层的方式显示节点的信息，所以在添加新的节点时应该注意它的父节点。使用 Nodes 属性的如下四种方法可以对节点进行操作：

- Add（）方法：增加新的节点。
- Remove（）方法：删除指定的节点。
- RemoveAt（）方法：删除指定的节点。
- Clear（）方法：删除所有的节点。

如果单击一个节点，则会引发 AfterSelect 事件。响应事件后，会在事件处理方法中编写单击节点后所要执行的代码。在事件处理方法中，TreeViewEventArgs 类参数用于确定用户单击了哪个节点。

另外，TreeView 内还有 MouseDown 事件和 MouseUp 事件。在事件处理方法中使用 TreeViewEventArgs 类参数来获取单击处点“Point”的 X 和 Y 的具体坐标值，然后使用 GetNodeAt（）方法来确定用户单击了哪个节点。

15.4 使用窗体菜单

菜单是窗体应用中的重要组成元素之一，在窗体内编写菜单程序，可以实现系统项目的特定功能。例如实现和 Office 样式类似的菜单。现实中常用菜单具有如下两种类型。

- 主菜单：显示在窗体标题下面的菜单，只有窗体才能有主菜单。
- 上下文菜单：和上下文操作相关的菜单，通常在窗体的相应位置单击鼠标右键来显示上下文菜单。

15.4.1 主菜单实现控件介绍

在.NET 2.0 框架被推出前，通常使用 MainMeue 控件来实现主菜单。在.NET 2.0 框架以后，使用 MaueStrip 控件来实现主菜单。在下面的内容中，将对上述控件的基本使用知识进

行简要介绍。

1．MainMeue 控件

MainMeue 控件是用来表示窗体菜单结构的容器，菜单是由表示单个菜单命令的 MeueItem 对象构成的。每个 MeueItem 可以是应用程序的命令，也可以是其他子菜单项的父菜单。如果将某 Form 对象的 Menu 属性设置为 MainMeue 对象，则该 MainMeue 对象就成为了 Form 对象的主菜单。

2．MaueStrip 控件

MaueStrip 控件是 Visual Studio 2010 工具箱内的重要控件工具之一，它可以灵活地在窗体创建主菜单。MaueStrip 控件支持多文档界面和菜单合并、工具提示和溢出。开发人员可以添加访问键、快捷键和选中标记等元素来增强菜单的可用性和可读性。

MaueStrip 的每一个菜单项都是一个 ToolStripMeueItem 对象，并且每个 ToolStripMeueItem 可以是应用程序的命令，也可以是其他子菜单项的父菜单，这样就构成了 MaueStrip 菜单的嵌套结构。

其中的 DropDownItems 属性包含了其子菜单项的 ToolStripMeueItem 对象集合，该集合主要通过如下四种方法实现对菜单项的操作。

- ❑ 方法 Add：添加新的子菜单项。
- ❑ 方法 Remove：删除某子菜单项。
- ❑ 方法 RemoveAt：删除某子菜单项。
- ❑ 方法 Clear：删除全部子菜单项。

ToolStripMeueItem 通过自身的属性可以设置菜单的外观和功能，各属性的具体说明如下。

- ❑ Checked：显示菜单被选中。
- ❑ CheckOnClick：菜单每次被单击后都更改选中状态的标记。
- ❑ Image：设置菜单的图像。
- ❑ ShortcutKeys：设置菜单的快捷键。

设置菜单访问键的方法比较简单，先为 ToolStripMeueItem 对象设置 Text 属性，然后为它加上下画线以作为访问键的字母前输入一个“&”符号即可。例如，输入“打开（&O）”作为某菜单项的 Text 属性，即可使“打开（&O）”作为菜单命令的访问键。

15.4.2 使用上下文菜单控件

上下文菜单控件即 ContextMenuStrip 控件，能够实现类似快捷键样式的上下文菜单效果。ContextMenuStrip 控件中的每一个菜单项也都是一个 ToolStripMeueItem 对象，并且每个 ToolStripMeueItem 也可以是应用程序的命令，也可以是其他子菜单项的父菜单，这样就构成了 MaueStrip 菜单的嵌套结构。

其中的 DropDownItems 属性包含了其子菜单项的 ToolStripMeueItem 对象集合，该集合主要通过如下四种方法实现对菜单项的操作。

- ❑ 方法 Add：添加新的子菜单项。
- ❑ 方法 Remove：删除某子菜单项。
- ❑ 方法 RemoveAt：删除某子菜单项。

❑ 方法 Clear：删除全部子菜单项。

MaueStrip 控件属性也适用于 ContextMenuStrip 控件，并且两者的使用方式也完全相同。为节省篇幅，对 ContextMenuStrip 的具体使用将不做详细介绍。

15.5 工具栏应用

工具栏是窗体应用中的重要组成元素之一，在窗体内编写工具栏程序，可以实现系统项目的特定功能。例如可以实现类似于 Visual Studio 2010 和 Office 样式的工具栏，如图 15−38 所示。

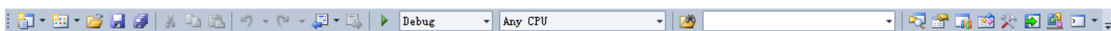


图 15−38 Visual Studio 2010 的工具栏

在 Visual Studio 2010 窗体程序开发应用中，使用 ToolStrip 控件来实现工具栏效果。在本节的内容中，将对 ToolStrip 控件的基本知识进行详细介绍。

15.5.1 ToolStrip 控件介绍

ToolStrip 控件的主要功能如下。

❑ 创建自定义格式的工具栏，并将工具栏项目实现排列和布局处理。

❑ 支持操作系统的典型外观和行为。

❑ 对所有的容器和包含的事件进行一致性处理，处理的方式和其他空间相同。

❑ 可以将一个 ToolStrip 控件拖到另一个 ToolStrip 控件内。

ToolStrip 控件使用其本身的属性、方法和事件来实现工具栏处理，其常用属性的信息如表 15−9 所示。

表 15−9 ToolStrip 常用属性信息

属　性	说　明
GripStyle	设置四个排列点是否在最左边
LayoutStyle	设置工具栏项的显示方式，默认为水平
Items	包含工具栏中所有项的集合
ShowItemToolTip	设置是否显示对应项的提示
Raft	指定包含的 ToolStrip 容器
Stretch	如果值为 True，工具栏会占据容器的总长
Dock	设置 ToolStrip 停靠在容器的哪一边
AllowItemReorder	设置拖放和排序是否专门由 ToolStrip 类进行
LayoutStyle	设置 ToolStrip 如何布局
Overflow	设置将 ToolStripItem 附加到哪里
IsDropDown	设置单击 ToolStrip 选项时是否显示下拉菜单中的其他项
OverflowButton	获取 ToolStripRenderer，用于启用溢出按钮
Renderer	获取 ToolStripRenderer，用于自定义 ToolStrip 的外观

在 ToolStrip 控件中可以使用其他控件和组件，除了本书前面介绍的控件可以被使用外，工具栏中还可以包含如下控件。

- ToolStripButton：按钮控件。
- ToolStripLable：标签控件。
- ToolStripSplitButton：右端具有下拉按钮的按钮。
- ToolStripDropDownButton：和 ToolStripSplitButton 类似，但是没有下拉按钮。
- ToolStripComboBox：组合框。
- ToolStripProgressBar：可以在工具栏上嵌入一个进度条。
- ToolStripSeparator：创建水平或垂直分隔符。
- ToolStripTextBox：显示一个文本框。

15.5.2　使用 ToolStrip 控件

下面将通过一个具体的实例来说明 ToolStrip 控件的具体使用过程。

实例 75：在窗体内实现一个指定样式的工具栏

本实例保存在“光盘：\daima\15”文件夹内，项目名为 gongju1。本实例的功能是在窗体内实现一个指定样式的工具栏效果。本实例的具体实现流程如下：

1）在 Visual Studio 2010 中新建一个名为“gongju1”的 Windows 应用程序。

2）从工具箱中拖入一个 ToolStrip 控件，设置属性 Text 值为“toolStrip1”，属性 name 为“toolStrip1”。

3）单击工具栏上的提示按钮并选择“Button”选项后，为工具栏添加一个按钮，如图 15-39 所示。

4）选择上面添加的按钮，鼠标右键单击在弹出对话框内选择“设置图像”命令，为此按钮设置素材图像，如图 15-40 所示。

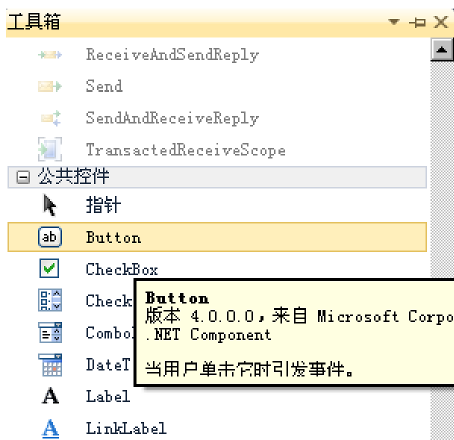


图 15-39　生成的菜单项



图 15-40　设置图像

5）按照上述步骤 3）和 4）继续在工具栏内添加多个按钮。

6）在新添加工具栏按钮时，选择“Separator”选项添加一条分割线，如图 15-41 所示。

7）按照上述步骤继续添加新按钮和分割线。

8）增加窗体的宽度，然后单击工具栏提示按钮并选择“ComboBox”选项，设置 name 为“toolStripComboBox1”，如图 15-42 所示。

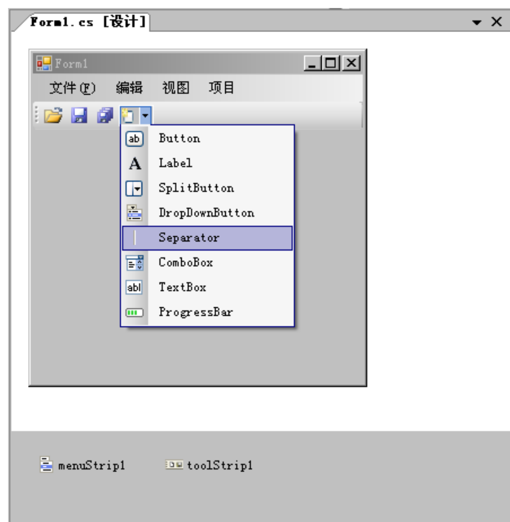


图 15-41　添加分割线

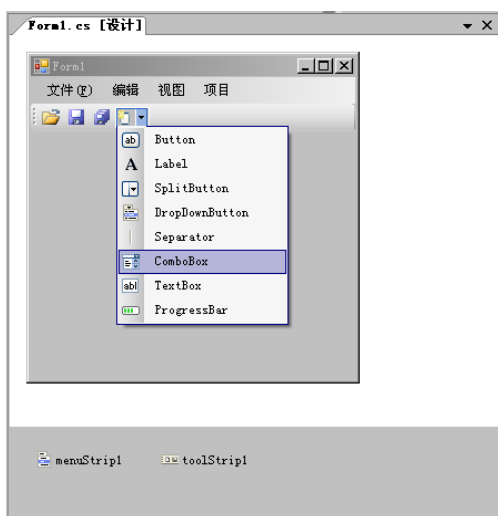


图 15-42　分组处理

9）按照上述步骤 8 继续在工具栏内添加 1 个 ComboBox，设置 name 为“toolStripComboBox2”。

10）分别设置上面添加 ComboBox 的 Items 属性，在“字符串集合编辑器”中分别输入对应的字符串，分别如图 15-43 和图 15-44 所示。

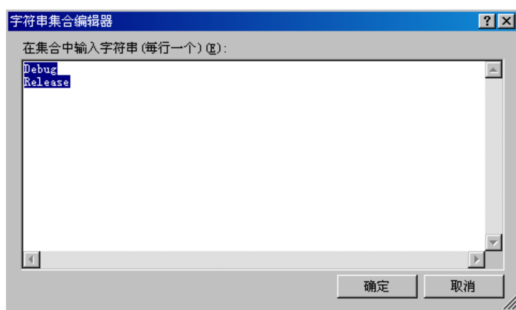


图 15-43　添加字符串

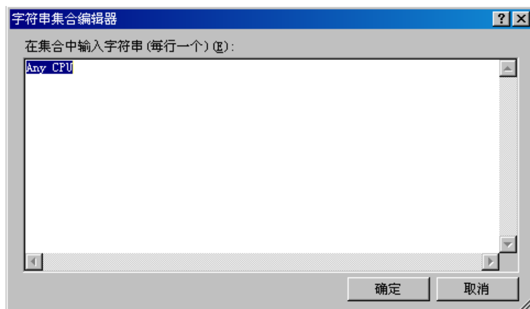


图 15-44　添加字符串

11）在自动生成的文件 Form1.cs 内添加窗体的 Load 事件代码，具体代码如下：

```
    private void Form1_Load(object sender, EventArgs e)
           {
              //设置默认选择的项
              toolStripComboBox1.SelectedIndex = 0;
              //设置默认选择的项
```

```
            toolStripComboBox2.SelectedIndex = 0;
        }
```

12）按照上述步骤继续添加新的工具栏项和对应的子项，为 ComboBox 左侧的按钮编写单击事件处理程序，具体代码如下：

```
    private void toolStripButton9_Click(object sender, EventArgs e)
        {
            //组合字符串 s
            string s1="配置："+toolStripComboBox1.SelectedItem.ToString();
            string s2="CPU 优化："+toolStripComboBox2.SelectedItem.ToString();
            string s = s1 + "  " + s2;
            //更改标题为字符串 s
            Text = s;
        }
```

13）按照上述步骤操作完毕后将项目文件保存，项目窗体的最终设计界面如图 15-45 所示。

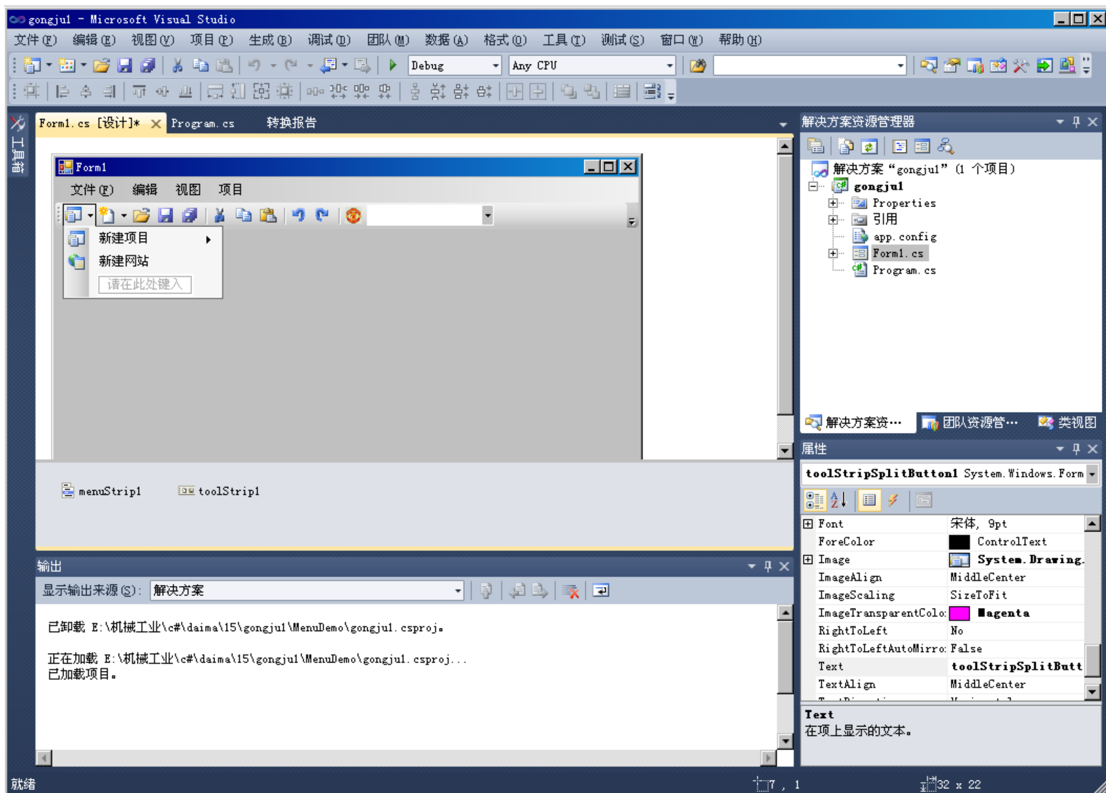


图 15-45　最终窗体设计界面

当程序执行后会显示一个具有指定菜单项的窗体，具体如图 15-46 所示；当单击对应菜单项后会显示对应的子项信息，如图 15-47 所示。

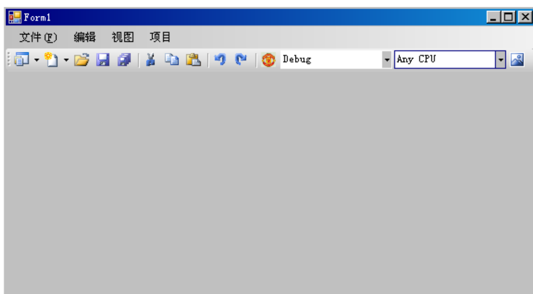


图 15-46　窗体运行效果

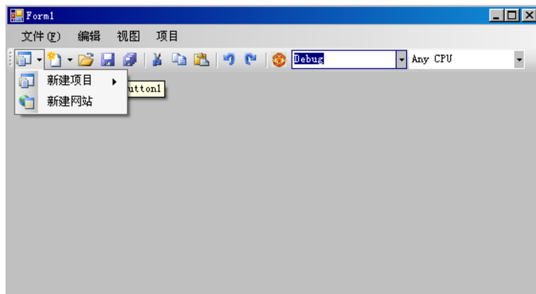


图 15-47　工具栏项效果

在项目文件中，文件 Program.cs 是项目的入口程序，主要实现代码如下：

```
    static class Program
    {
        /// <summary>
        /// 应用程序的主入口点。
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new Form1());
        }
    }
```

控件属性的设置代码在文件 Form1.Designer.cs 内，主要代码格式如下：

```
      ……………………………………………………………………
            // menuStrip1
            this.menuStrip1.Items.AddRange(new System.Windows.Forms. To
            olStripItem[] {
            this.文件 ToolStripMenuItem,
            this.编辑 ToolStripMenuItem,
            this.视图 ToolStripMenuItem,
            this.项目 ToolStripMenuItem});
            this.menuStrip1.Location = new System.Drawing.Point(0, 0);
            this.menuStrip1.Name = "menuStrip1";
            this.menuStrip1.Size = new System.Drawing.Size(556, 24);
            this.menuStrip1.TabIndex = 0;
            this.menuStrip1.Text = "menuStrip1";
            // 文件 ToolStripMenuItem
            this.文件 ToolStripMenuItem.DropDownItems.AddRange(new System.
            Windows.Forms.ToolStripItem[] {
            this.新建 ToolStripMenuItem,
            this.打开 ToolStripMenuItem,
```

```
            this.toolStripSeparator1,
            this.添加 ToolStripMenuItem,
            this.toolStripSeparator2,
            this.关闭 ToolStripMenuItem,
            this.关闭项目 ToolStripMenuItem,
            this.toolStripSeparator3,
            this.保存 ToolStripMenuItem,
            this.全部保存 ToolStripMenuItem});
            this.文件 ToolStripMenuItem.Name = "文件 ToolStripMenuItem";
            this.文件 ToolStripMenuItem.Size=new System.Drawing.Size(59,20);
            this.文件 ToolStripMenuItem.Text = "文件(&F)";
            // 新建 ToolStripMenuItem
            this.新建 ToolStripMenuItem.DropDownItems.AddRange(new System.
            Windows.Forms.ToolStripItem[] {
            this.项目 ToolStripMenuItem1,
            this.网站 ToolStripMenuItem,
            this.文件 ToolStripMenuItem1});
            this.新建 ToolStripMenuItem.Name = "新建 ToolStripMenuItem";
            this.新建 ToolStripMenuItem.Size=new System.Drawing.Size(152,22);
            this.新建 ToolStripMenuItem.Text = "新建(&N)";
            // 项目 ToolStripMenuItem1
            this.项目 ToolStripMenuItem1.Name = "项目 ToolStripMenuItem1";
            this.项目 ToolStripMenuItem1.Size=new System.Drawing.Size(130,22);
            this.项目 ToolStripMenuItem1.Text = "项目(&P)...";
            // 网站 ToolStripMenuItem
            this.网站 ToolStripMenuItem.Name = "网站 ToolStripMenuItem";
            this.网站 ToolStripMenuItem.Size = new System.Drawing. Size
            (130, 22);
            this.网站 ToolStripMenuItem.Text = "网站(&W)...";
    ……………………………………………………
```

15.6　窗体对话框

对话框是窗体应用中的重要组成元素之一，在窗体内编写对话框程序，可以实现现实项目中的指定功能。在窗体项目程序开发应用中，可以自定义对话框，也可以使用 Visual Studio 2010 提供的专用控件来实现。

15.6.1　自定义实现对话框

对话框通常在需要时弹出，在不需要时被关闭，并且对话框不包含菜单栏和滚动条等元素，而只有最小化、最大化的按钮。可以将对话框看成一种特殊的 Form 窗体，根据自己的定义样式来设计独立的对话框程序。

实现自定义对话框效果的实现流程如下。

1）将窗体的 FormBorderStyle 属性设置为 FixedDialog，将 MinimsizeBox 和 MaximizeBox 属性设置为 false。

2）如果需要在对话框中访问父窗体的信息，需要将 ParentForm 属性返回的引用显式转换为父窗体的类型。

3）通过 VS 控件实现对话框的提示数据，并使用窗体设计器控制各控件的位置和大小。

4）通过专用的事件处理方法在对话框内显示处理的数据。

5）使用 ShowDialog 方法模式显示对话框，并等待用户的操作。并且可以在关闭时获取用户的输入数据，根据 ShowDialog 返回值确定用户单击的操作按钮。

实例 76：将 Form 窗体自定义为对话框

下面将通过一个具体的实例来说明将 Form 窗体自定义为对话框的具体实现过程。本实例保存在“光盘：\daima\15”文件夹内，项目名为 duihua1，功能是在窗体内实现对话框处理效果。本实例的具体实现流程如下：

1）在 Visual Studio 2010 中新建一个名为“duihua1”的 Windows 应用程序。

2）设置窗体的 name 属性为“Form1”，设置 Text 属性值为“chuangti1”。

3）从工具箱中拖入一个 Button 控件到 Form1 的顶部，设置 name 为“textBox1”，设置 Text 属性值为空。

4）从工具箱中拖入一个 TextBox 控件到 button1 底部，并设置其 Text 属性为空，设置 name 为“textBox1”。

5）在当前项目内再次添加一个名为“duihua”的窗体。

6）在设计器中选中窗体 duihua，设置 FormBorderStyle 属性值为“FixedDialog”，设置 MinimsizeBox 和 MaximizeBox 属性值为 false，设置 Text 属性值为“请输入信息”。

7）从工具箱中拖入一个 Lable 控件到 duihua 的顶部，设置 name 为“label1”，设置 Text 属性值为“姓名”。

8）从工具箱中拖入一个 Lable 控件到 label1 的下面，设置 name 为“label2”，设置 Text 属性值为“电话”。

9）分别拖入两个 TextBox 控件到 label1 和 label2 后面。

10）从工具箱中拖入一个 GroupBox 控件到 duihua 窗体中部，设置 name 为“groupBox1”，设置 Text 属性值为“性别”。

11）从工具箱中拖入两个 RadioButton 控件到 groupBox1 内，分别设置 name 为“radioButton1”和“radioButton2”，分别设置 Text 属性值为“男”和“女”。

12）从工具箱中拖入一个 GroupBox 控件到 duihua 窗体底部，设置 name 为“groupBox2”，设置 Text 属性值为“基本信息”。

13）从工具箱中拖入三个 CheckBox 控件到 groupBox2 内，分别设置 name 为“checkBox1”、“checkBox2”和“checkBox3”，分别设置 Text 属性值为“专科”、“本科”和“研究生”。

14）从工具箱中拖入两个 Button 控件到 dhuihua 窗体内，分别设置 name 为“button1”和“button2”，分别设置 Text 属性值为“确定”和“取消”。

15）分别选中上面的 button1 和 button2，在“属性”中的“事件”选项中双击 Click 事件，编写对应的事件处理代码。具体实现代码如下：

```
    private void button1_Click(object sender, EventArgs e)
```

```
        {
            // 确定
            this.DialogResult = DialogResult.OK;
            this.Close();
        }
        private void button2_Click(object sender, EventArgs e)
        {
            // 取消
            this.DialogResult = DialogResult.Cancel;
            this.Close();
        }
```

16）在自动生成的文件 Form1.cs 中，定义类 InfoDlg：

```
    public partial class InfoDlg : Form
    {
        // 姓名属性
        public string InputName
        {
            get
            {
                return textBox1.Text;
            }
        }
        // 电话属性
        public string Telephone
        {
            get
            {
                return textBox2.Text;
            }
        }
        // 性别属性
        public string Sex
        {
            get
            {
                if (radioButton1.Checked)
                    return "男";
                else
                    return "女";
            }
        }
        // 是否专科属性
        public bool zhuan
        {
            get
```

```
                {
                    return checkBox1.Checked;
                }
            }
            // 是否本科毕业
            public bool benke
            {
                get
                {
                    return checkBox2.Checked;
                }
            }
            // 是否研究生毕业
            public bool yanjiu
            {
                get
                {
                    return checkBox3.Checked;
                }
            }
            public InfoDlg()
            {
                InitializeComponent();
            }
```

17）选中 Form1 窗体上面的 button1，在“属性”中的“事件”选项中双击 Click 事件，编写获取对话框数据事件的处理代码。具体实现代码如下：

```
        private void button1_Click(object sender, EventArgs e)
            {
                // 创建对话框
                InfoDlg dlg = new InfoDlg();
                // 显示对话框，并获取返回结果
                if (dlg.ShowDialog() == DialogResult.Cancel)    // 取消
                {
                    MessageBox.Show("用户取消了操作。", "Dialog Sample");
                    return;
                }
                // 确定，获取信息
                string info = "";
                info += "姓名：" + dlg.InputName + System.Environment.NewLine;
                info += "电话：" + dlg.Telephone + System.Environment.NewLine;
                info += "性别：" + dlg.Sex + System.Environment.NewLine;
                info += "是否专科毕业：" + dlg.zhuan + System.Environment.NewLine;
                info += "是否本科毕业：" + dlg.benke + System.Environment.NewLine;
                info += "是否研究生毕业：" + dlg.yanjiu;
                // 在文本框中显示获取的信息
```

```
            this.textBox1.Text = info;
        }
```

18）经过上述操作处理后，整个项目设计完毕。完成后的具体设计界面分别如图 15-48 和图 15-49 所示。

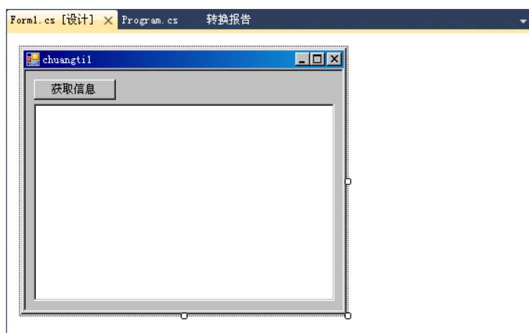


图 15-48　Form1 窗体设计界面结构

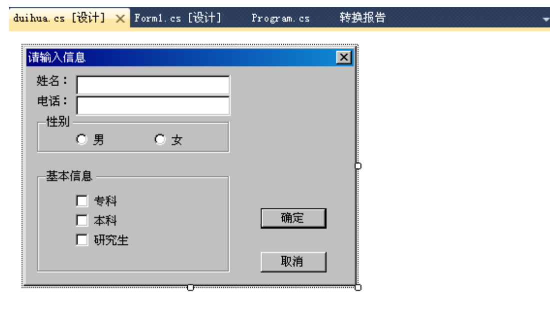


图 15-49　duihua 窗体设计界面结构

当上述项目运行后，会首先按照指定的样式显示窗体的各个元素，如图 15-50 所示。当单击“获取信息”按钮后，会弹出“输入信息”对话框，如图 15-51 所示。

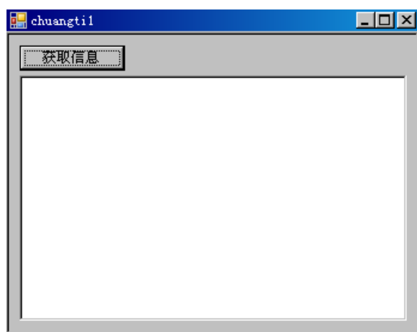


图 15-50　窗体界面效果

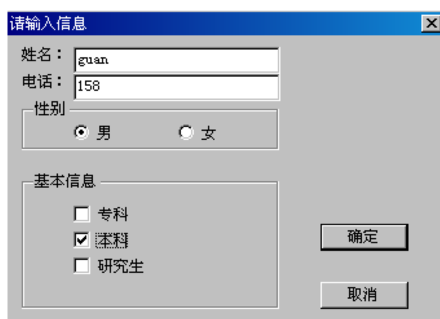


图 15-51　“输入信息”对话框

当输入并选择信息，单击“确定”按钮后，会在窗体内获取显示用户输入的信息，如图 15-52 所示。

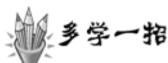


图 15-52　显示信息

至此，整个项目设计完毕。各控件的属性设置代码分别在文件 Form1.Designer.cs 和 duihua.Designer.cs 内，主要代码格式如下：

```
………………………………………
            // label1
            this.label1.AutoSize = true;
            this.label1.Location = new System.Drawing.Point(9, 8);
            this.label1.Margin = new System.Windows.Forms.Padding(2, 0,
            2, 0);
            this.label1.Name = "label1";
            this.label1.Size = new System.Drawing.Size(41, 12);
            this.label1.TabIndex = 0;
            this.label1.Text = "姓名：";
………………………………………
            // InfoDlg
            this.AcceptButton = this.button1;
            this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 12F);
            this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
            this.CancelButton = this.button2;
            this.ClientSize = new System.Drawing.Size(338, 222);
            this.Controls.Add(this.button2);
            this.Controls.Add(this.button1);
            this.Controls.Add(this.groupBox2);
            this.Controls.Add(this.groupBox1);
            this.Controls.Add(this.textBox2);
            this.Controls.Add(this.label2);
            this.Controls.Add(this.textBox1);
            this.Controls.Add(this.label1);
            this.FormBorderStyle = System.Windows.Forms.FormBorderStyle.
            FixedDialog;
            this.Margin = new System.Windows.Forms.Padding(2, 2, 2, 2);
            this.MaximizeBox = false;
            this.MinimizeBox = false;
            this.Name = "InfoDlg";
            this.StartPosition = System.Windows.Forms.FormStartPosition.
            CenterParent;
            this.Text = "请输入信息";
            this.Load += new System.EventHandler(this.InfoDlg_Load);
            this.groupBox1.ResumeLayout(false);
            this.groupBox1.PerformLayout();
            this.groupBox2.ResumeLayout(false);
            this.groupBox2.PerformLayout();
            this.ResumeLayout(false);
            this.PerformLayout();
………………………………………
```

事件处理代码在文件 Form1.cs 和 duihua.cs 内，而文件 Program.cs 是项目的入口程序。



在默认情况下，用一个按钮来触发一个事件，从而达到实现某一功能的操作，例如按下一个按钮可以删除数据库中某一记录或某一批记录等。通常要进行这些操作前都先要进行提示确认的步骤，这个提示步骤是：当用户进行某一操作时，首先弹出一个操作确认选择对话框，用户确认后，操作才能有效执行，否则中止操作。这样做既可以给用户提供一个很好的操作界面，也防止了误操作的发生。

15.6.2　使用对话框控件

在.NET 框架中，定义了专用的对话框控件实现对话框效果。类 CommonDiaolog 是各对话框类的基类，它通过本身的方法和事件来实现对话框处理。类 CommonDiaolog 的常用方法和事件如下。

- 方法 ShowDialog：显示一个通用的对话框。
- 方法 Reset：每个派生的对话框类都执行 Reset 方法，能够把对话框类的所有属性设置为它的默认值。
- HelpRequest 事件：单击对话框上的“Help”按钮时会激活这个事件。

.NET 框架中主要对话框类的具体说明如下。

- 类 OpenFileDialog：用户可以选择和浏览要打开的文件，并可以设置打开多个文件。
- 类 SaveFileDialog：用户可以为保存的文件指定保存路径。
- 类 PrintDialog：设置选择打印机，并设置打印选项。
- PageSetupDialog：配置页面的边框。
- PrintPreviewDialog：在屏幕上进行打印预览。
- FontDialog：列出所有已安装的 Windows 字体、样式和字号，并写显示对应的预览效果。
- ColorDialog：用于选择颜色。
- FolderBrowserDialog：用于选择或创建目录。

上述对话框控件都在工具箱内，开发人员可以迅速使用。

实例 77：使用对话框控件实现对话框处理效果

下面将通过一个具体的实例来说明对话框控件的具体使用过程。本实例保存在“光盘：\daima\15”文件夹内，项目名为 duihua2，功能是在窗体内实现对话框处理效果。本实例的具体实现流程如下：

1）在 Visual Studio 2010 中新建一个名为“duihua2”的 Windows 应用程序。

2）设置窗体的 name 属性值为“Form1”，设置属性 Text 值为“对话框控件”。

3）从工具箱中拖入一个 TextBox 控件，并设置其 Text 属性值为空，设置 name 为“textBox1”，设置 ScrollBars 属性值为“Both”，设置 MultiLine 为“True”。

4）在自动生成的文件 Form1.cs 中，定义构造函数 Form1()，并使用 CreateMenuStrip()为窗体创建 2 个主菜单“文件”和“帮助”，并在“文件”下设置“打开”、“另存为”和“退出”三个子菜单，在“帮助”下设置“关于”子菜单。

上述功能的具体实现代码如下：

```
public Form1()
        {
            InitializeComponent();
            CreateMenuStrip();
        }
        private void CreateMenuStrip()
        {
            // 主菜单
            MenuStrip menu = new MenuStrip();
            // 文件菜单
            ToolStripMenuItem mi1 = new ToolStripMenuItem("文件(&F)");
            // 文件菜单里的菜单项
            ToolStripMenuItem mi11= new ToolStripMenuItem("打开(&O)...");
            ToolStripMenuItem mi12=new ToolStripMenuItem("另存为(&A)...");
            ToolStripMenuItem mi13 = new ToolStripMenuItem("退出(&X)");
            // 连接文件菜单里的菜单项的Click事件处理程序
            mi11.Click += new EventHandler(mi11_Click);
            mi12.Click += new EventHandler(mi12_Click);
            mi13.Click += new EventHandler(mi13_Click);
            // 将菜单项添加到文件菜单中
            mi1.DropDownItems.Add(mi11);
            mi1.DropDownItems.Add(mi12);
            mi1.DropDownItems.Add(mi13);
            // 帮助菜单
            ToolStripMenuItem mi2 = new ToolStripMenuItem("帮助(&H)");
            // 帮助菜单里的菜单项
            ToolStripMenuItem mi21=new ToolStripMenuItem("关于(&A)...");
            // 连接帮助菜单里的菜单项的Click事件处理程序
            mi21.Click += new EventHandler(mi21_Click);
            // 将菜单项添加到帮助菜单中
            mi2.DropDownItems.Add(mi21);
            // 将菜单项添加到主菜单中
            menu.Items.Add(mi1);
            menu.Items.Add(mi2);
            // 将菜单项添加窗体中
            this.Controls.Add(menu);
            // 将菜单设置为窗体的主菜单
            this.MainMenuStrip = menu;
        }
```

5）为子菜单“打开”设置 Click 事件的处理方法 mi11_Click，具体代码如下：

```
private void mi11_Click(object sender, EventArgs e)
        {
            // 打开文件
            // 创建OpenFileDialog对象
            OpenFileDialog dlg = new OpenFileDialog();
```

```
            // 设置 OpenFileDialog 对象的属性
            dlg.InitialDirectory = "c:\\";
            dlg.Filter = "txt files (*.txt)|*.txt|All files (*.*)|*.*";
            dlg.FilterIndex = 0;
            // 显示对话框并获取返回结果
            if (dlg.ShowDialog() == DialogResult.OK)
            {
                // 读取文件
                string s = "";
                StreamReader sr = new StreamReader(dlg.FileName);
                while (!sr.EndOfStream)
                {
                    s += sr.ReadLine() + System.Environment.NewLine;
                }
                sr.Close();
                // 在 textBox1 中显示
                this.textBox1.Text = s;
            }
        }
```

6）为子菜单“另存为”设置 Click 事件的处理方法 mi12_Click，具体代码如下：

```
    private void mi12_Click(object sender, EventArgs e)
        {
            // 保存文件
            // 创建 SaveFileDialog 对象
            SaveFileDialog dlg = new SaveFileDialog();
            // 设置 SaveFileDialog 对象的属性
            dlg.InitialDirectory = "c:\\";
            dlg.Filter = "txt files (*.txt)|*.txt|All files (*.*)|*.*";
            dlg.FilterIndex = 0;
            // 显示对话框并获取返回结果
            if (dlg.ShowDialog() == DialogResult.OK)
            {
                // 写入文件
                StreamWriter sw = new StreamWriter(dlg.FileName);
                sw.Write(this.textBox1.Text);
                sw.Close();
            }
        }
```

7）为子菜单“退出”和“关于”分别设置 Click 事件的处理方法 mi13_Click 和 mi21_Click，具体代码如下：

```
    private void mi13_Click(object sender, EventArgs e)
    {
            this.Close();
```

```
            }
            private void mi21_Click(object sender, EventArgs e)
            {
                MessageBox.Show("关于 OpenFileDialog 和 SaveFileDialog 的示例。",
                "FileDialog Sample");
            }
            private void Form1_Load(object sender, EventArgs e)
            {
            }
```

8）经过上述操作处理后，整个项目设计完毕。完成后的具体设计界面如图 15-53 所示。

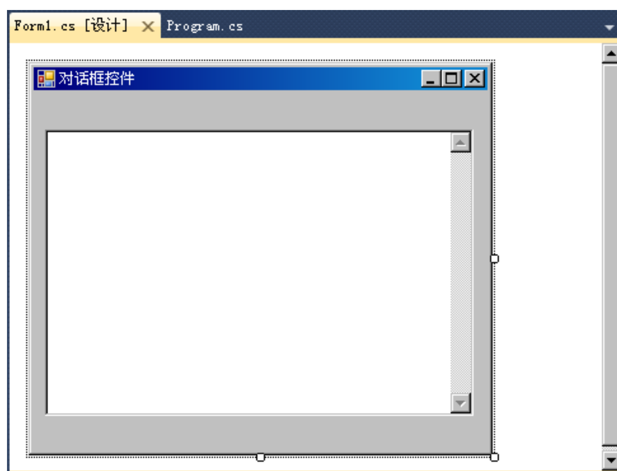


图 15-53　窗体设计界面结构

当上述项目运行后，将按照指定的样式显示窗体的各个元素，如图 15-54 所示。当单击“文件”菜单下的“打开”子菜单后，将弹出“文件打开”对话框，如图 15-55 所示。

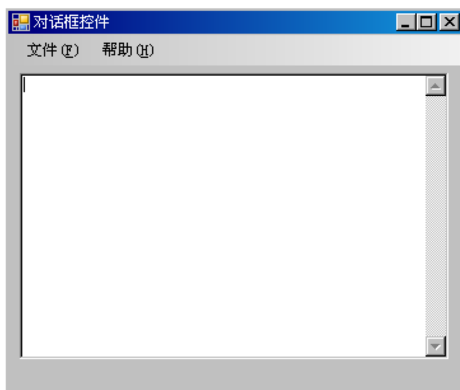


图 15-54　窗体界面效果

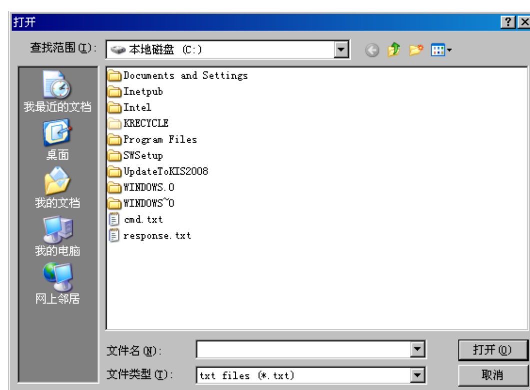


图 15-55　“文件打开”对话框

当单击“文件”菜单下的“另存为”子菜单后，将弹出“文件保存”对话框，如图 15-56 所示。

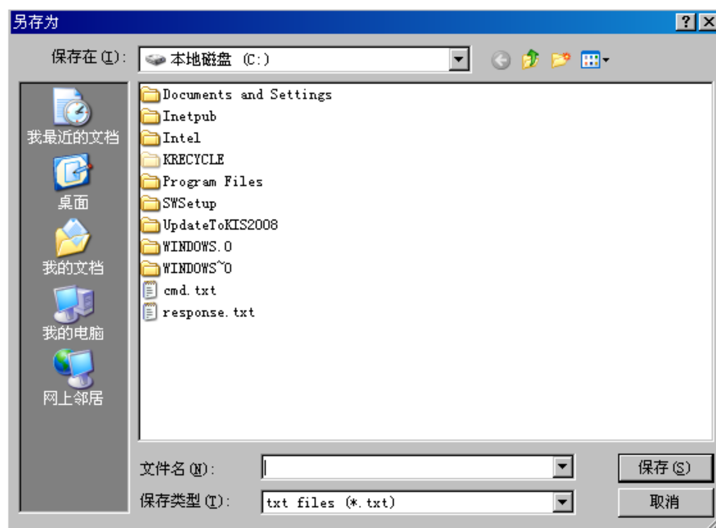


图 15-56　“文件保存”对话框

至此，整个项目设计完毕。项目内各控件的属性设置代码在文件 Form1.Designer.cs 内，具体的代码格式如下：

```
    ..................................................
            // textBox1
            this.textBox1.Anchor = ((System.Windows.Forms.AnchorStyles)
             ((((System.Windows.Forms.AnchorStyles.Top | System.Windows.
            Forms.AnchorStyles.Bottom)
                        | System.Windows.Forms.AnchorStyles.Left)
                        | System.Windows.Forms.AnchorStyles.Right)));
            this.textBox1.Location = new System.Drawing.Point(9, 30);
            this.textBox1.Margin = new System.Windows.Forms.Padding(2);
            this.textBox1.Multiline = true;
            this.textBox1.Name = "textBox1";
            this.textBox1.ScrollBars = System.Windows.Forms.ScrollBars.
            Both;
            this.textBox1.Size = new System.Drawing.Size(337, 225);
            this.textBox1.TabIndex = 1;
            // Form1
            this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 12F);
            this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
            this.ClientSize = new System.Drawing.Size(357, 282);
            this.Controls.Add(this.textBox1);
            this.Margin = new System.Windows.Forms.Padding(2);
            this.Name = "Form1";
            this.Text = "对话框控件";
            this.Load += new System.EventHandler(this.Form1_Load);
            this.ResumeLayout(false);
            this.PerformLayout();
```

事件处理代码在文件 Form1.cs 和 duihua.cs 内，而文件 Program.cs 是项目的入口程序。


多学一招

上述实例中，在窗体内实现了简单的对话框处理效果。经过简单总结，C#中的常用对话框控件有如下六种。

1）文件对话框(FileDialog) ，最常用的有如下两个。

- 打开文件对话框(OpenFileDialog)
- 保存文件对话(SaveFileDialog)

2）字体对话框(FontDialog)

3）颜色对话框(ColorDialog)

4）打印预览对话框(PrintPreviewDialog)

5）页面设置(PrintDialog)

6）打印对话框(PrintDialog)

15.7 疑难问题解析

本章详细介绍了 C#中 Windows 窗体编程的基本知识。本节将对本章中比较难以理解的问题进行讲解。

读者疑问：请举例介绍 ListBox 和 ComboBox 控件的使用方法？

解答：下面将通过一个具体的实例来说明，ListBox 和 ComboBox 控件的综合使用过程。本实例保存在“15”文件夹内，项目名为 Listchuli，功能是为窗体设置列表框选项。本实例的具体实现流程如下:

1）新建一个名为“Listchuli”的 Windows 应用程序。

2）从工具箱中拖入到窗体顶部一个 ListBox 控件，设置 name 为“listBox1”，并调整它的大小。

3）选中上述 listBox1 控件，在“属性”的“事件”界面双击 SelectedIndexChanged 事件。

4）从工具箱中分别拖入到 listBox1 右侧三个 Button 控件，从上到下依次进行如下设置。

- 第一个 Button：设置属性 name 值为“button1”，设置 Text 属性为“增加”。
- 第二个 Button：设置属性 name 值为“button2”，设置 Text 属性为“移出”。
- 第三个 Button：设置属性 name 值为“button3”，设置 Text 属性为“全部删除”。

5）分别选中上述各 Button 控件，在“属性”的“事件”界面双击 Click 事件。

6）从工具箱中拖入一个 CheckBox 控件到 listBox1 的下面，设置 name 为“checkBox1”，Text 属性值为“可以多项选择”。

7）在 Form1.cs 中增加如下事件处理代码:

```
    private void listBox1_SelectedIndexChanged(object sender,EventArgs e)
```

```
{
    string indices = "";
    foreach (int index in listBox1.SelectedIndices)
    {
        indices += index.ToString() + " ";
    }
    MessageBox.Show("选择项: " + indices, "List Sample");
}
private void button1_Click(object sender, EventArgs e)
{
    listBox1.Items.Add(DateTime.Now.ToString());
}
private void button2_Click(object sender, EventArgs e)
{
    int count = listBox1.SelectedIndices.Count;
    for (int i = count - 1; i >= 0; --i)
        listBox1.Items.RemoveAt(listBox1.SelectedIndices[i]);
}
private void button3_Click(object sender, EventArgs e)
{
    listBox1.Items.Clear();
}
private void checkBox1_CheckedChanged(object sender,EventArgs e)
{
    if (checkBox1.Checked)
    {
        MessageBox.Show("您选择了多项选择。", "List Sample");
        listBox1.SelectionMode = SelectionMode.MultiSimple;
    }
    else
    {
        MessageBox.Show("您取消选择了多项选择。", "List Sample");
        listBox1.SelectionMode = SelectionMode.One;
    }
}
```

8）从工具箱中拖入到窗体底部一个 ComboBox 控件，设置 name 为“comboBox1”，并调整它的大小。

9）选中上述 comboBox1 控件，在“属性”的“事件”界面双击 SelectedIndexChanged 事件。

10）从工具箱中拖入到窗体底部右侧 1 个 Button 控件，设置属性 name 值为“button4”，设置 Text 属性为“填充”。

11）选中上述 button4 控件，在“属性”的“事件”界面双击 Click 事件，并在 Form1.cs 中为上述控件添加对应的事件处理方法，具体代码如下:

```
    private void button4_Click(object sender, EventArgs e)
            {
                comboBox1.Sorted = true;
                Random rnd = new Random();
                for (int i=0; i<10; ++i)
                    comboBox1.Items.Add(rnd.Next());
            }
            private void comboBox1_SelectedIndexChanged(object sender,EventArgs e)
            {
                MessageBox.Show("选择项:"+comboBox1.SelectedIndex, "List Sample");
            }
```

12）经过上述操作处理后，整个项目设计完毕。完成后的具体设计界面如图 15-57 所示。

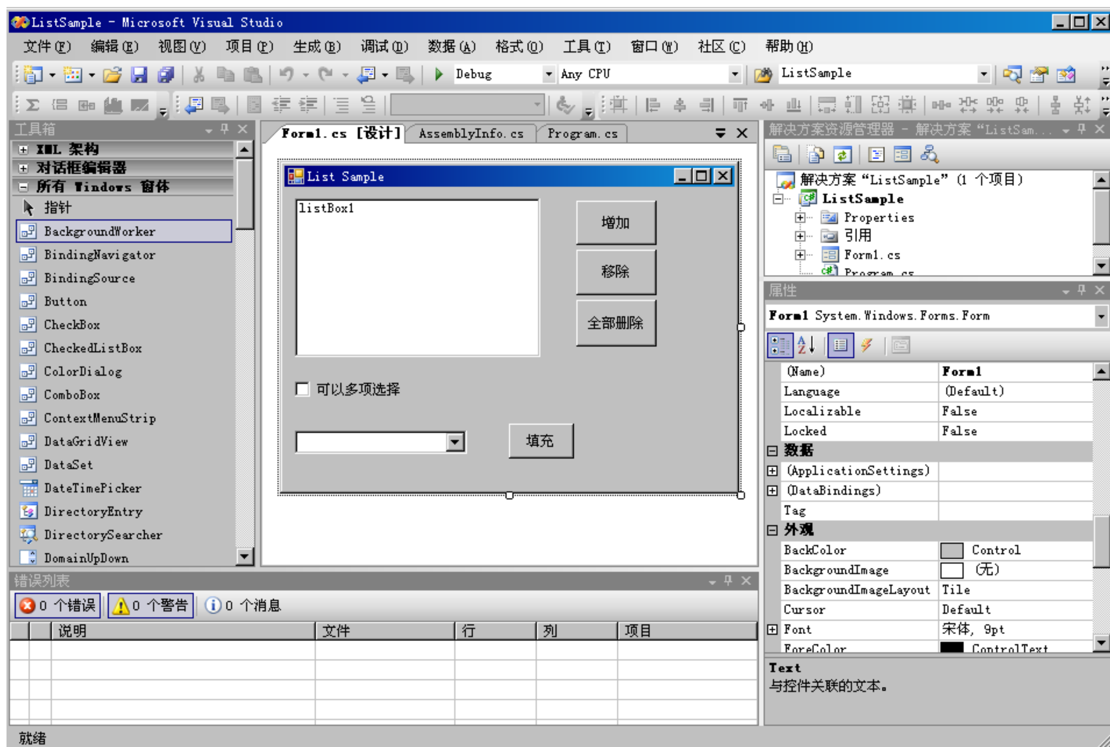


图 15-57　项目设计界面结构

当上述项目运行后，会按照指定的样式显示窗体的各个元素，如图 15-58 所示。当单击“增加”按钮或“移除”按钮后，会在列表内显示对应处理后的数据，如图 15-59 所示。单击“全部删除”按钮后，会将列表内的数据清除。单击“填充”按钮后会在 ComboBox 框内显示对应的数据，如图 15-60 所示。当选中框内的某选项后会弹出对应的对话框信息，如图 15-61 所示。

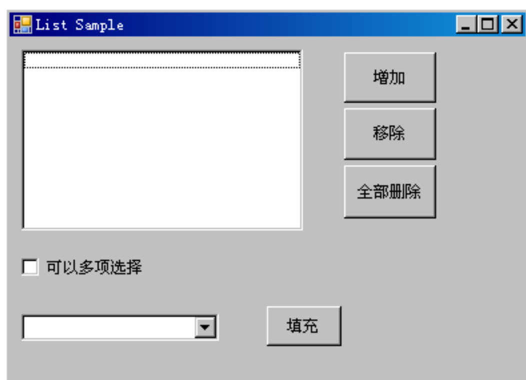


图 15-58　窗体界面效果



图 15-59　列表内显示增加/移除后的数据



图 15-60　填充数据

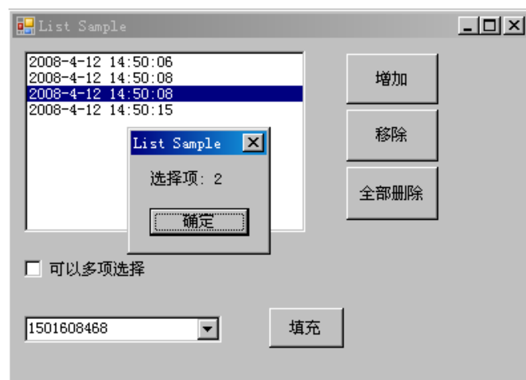


图 15-61　对话框信息

至此，整个项目设计完毕。各控件的属性设置代码在文件 Form1.Designer.cs 内，主要代码如下：

```
……………………………………
this.button3.Location = new System.Drawing.Point(251, 96);
          this.button3.Margin = new System.Windows.Forms.Padding(2);
          this.button3.Name = "button3";
          this.button3.Size = new System.Drawing.Size(69, 40);
          this.button3.TabIndex = 4;
          this.button3.Text = "全部删除";
          this.button3.UseVisualStyleBackColor = true;
          this.button3.Click+=new  System.EventHandler(this.button3_Click);
          // checkBox1
……………………………………
          this.checkBox1.AutoSize = true;
          this.checkBox1.Location = new System.Drawing.Point(9, 165);
          this.checkBox1.Margin = new System.Windows.Forms.Padding(2);
          this.checkBox1.Name = "checkBox1";
          this.checkBox1.Size = new System.Drawing.Size(72, 16);
```

```
                this.checkBox1.TabIndex = 5;
                this.checkBox1.Text = "可以多项选择";
                this.checkBox1.UseVisualStyleBackColor = true;
                this.checkBox1.CheckedChanged+=new System.EventHandler(this.
                checkBox1_CheckedChanged);
                //
                // comboBox1
                this.comboBox1.FormattingEnabled = true;
                this.comboBox1.Location = new System.Drawing.Point(9, 208);
                this.comboBox1.Margin = new System.Windows.Forms.Padding(2);
                this.comboBox1.Name = "comboBox1";
                this.comboBox1.Size = new System.Drawing.Size(148, 20);
                this.comboBox1.TabIndex = 6;
                this.comboBox1.SelectedIndexChanged += new System.EventHan
                dler(this.comboBox1_SelectedIndexChanged);
        ……………………………………
```

事件处理代码在文件 Form1.cs 内，而文件 Program.cs 是项目的入口程序。



职场点拨——初次见面要留下好印象

在职场中会不可避免地要面见一些客户，即使是作为一名程序员也不例外。无论约见客户的目的是什么，都要给客户留下一个好的第一印象。笔者在此给出几点建议。

（1）提前起程，不能迟到

准时守信是商场法则。在和客户进行第一次约见时，要充分做到防止遇到修路、塞车、天气转坏等意外发生，所以要提前起程，以免迟到。若早到，可以跟接待人员说："我是××，跟××有约，我早到了几分钟。"但是一定不要迟到，否则会带给客户不好的印象。

（2）事先弄清客户姓名。

可以先问客户公司的其他人或是直接问客户本人，不要怕问对方，他会因此对你印象深刻，觉得你很有礼貌。如果这名字不好念，就多练习几遍，直至熟悉为止。

（3）保持外表仪容

最好穿正装以表达对客户的重视，保持良好的外表和仪容是进行业任何务洽谈的基础。

（4）准备好给客户的资料

第一次和客户洽谈，要为他准备好完整的资料。包括本公司的简介、项目需求分析、售后、造价预算等，而且必须整理好，内容吸引人且适用。

（5）笑脸迎接客户

笑容是面对客户的第一法则。山姆·沃尔顿要求沃尔玛的员工遵循"10 尺"定律。无论何时，顾客只要在 10 尺之内，就应该保持笑容，看着对方眼睛，准备亲切问好。要让客户觉得你真的很高兴能去拜访。

第 16 章　Web 应用编程

C#功能强大，除了能够编写 Windows 窗体程序外，还能够编写 Web 应用程序。当前互联网行业迅速发展，各种 Web 编程语言便应运而生。微软最新推出的 ASP.NET 以其优越的性能，迅速成为 Web 编程的最主流技术。而 ASP.NET 使用的最主流语言便是 C#，通过 ASP.NET 不仅能够编写出现实中常用的 Web 应用程序，并且能够使用它本身提供的多个控件来实现更加复杂的 Web 应用。在本章的内容中，将对 C#在 ASP.NET 中的应用知识进行简要介绍，并通过简单的实例来介绍其具体的使用流程。通过本章能学到如下知识。

- Web 编程概述。
- 创建一个简单的 ASP.NET 程序。
- ASP.NET 控件。
- 状态管理。
- ASP.NET 和数据库的连接。
- 职场点拨——有错要勇于面对同事。

2008 年 XX 月 XX 日，多云

不知不觉间已经工作两年了，我几乎每天都坐在办公室里。看着身边的同事，我愈发感觉同事关系很重要，暗下决心一定要和他们好好相处。



一问一答

小菜：“和同事相处真是一门学问，我一定要和他们好好的相处。”

Wisdom：“呵呵，只要你在职场中生活，就需要面对同事关系。同事能在技术上给你点拨，并且能帮助你成长。即使是在激烈的辩论中，也能使你意识到问题的所在，提高你的业务能力，所以你要好好处理同事关系。”

小菜：“嗯，同事关系确实很重要。言归正传，本章的 Web 编程很重要吗？”

Wisdom：“Web 编程是当前程序开发的一大领域，Web 是基于网络的。基于 C#的 ASP.NET 技术是当前 Web 开发的主流技术，本章将讲解这门技术的基本知识。”

16.1　Web 编程概述

Web 应用程序能够使 Web 服务器向客户端机器发送处理后的 HTML 代码，这些代码能够在客户的浏览器中显示。当用户在浏览器中输入字符串数据后，会通过 Web 应用程序获取页面上的数据，并将数据传递给服务器。Web 服务器将数据处理完成后，将处理结果以静

态 HTML 代码的格式回复给客户端用户。在上面的整个处理过程中，Web 应用程序起到了中间处理转换和数据传输的作用。使所有的处理工作交给 Web 服务器来完成，而客户端机器只需要使用浏览器将返回的处理结果显示出来即可。上述 Web 应用程序的具体处理流程如图 16−1 所示。

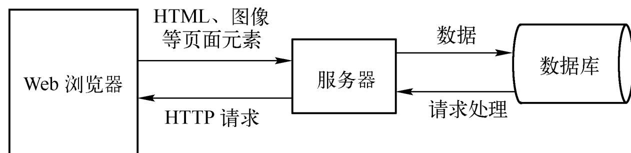


图 16−1　Web 程序处理流程图

C#完全支持最新的 ASP.NET 4.0，通过 ASP.NET 技术可以创建动态的 Web 程序。使用 ASP.NET 编写的 Web 页面和客户端的 Windows 程序类似，不但能够响应和处理 HTTP 请求，而且能够使用 TextBox 和 Bottom 等控件来迅速编写 ASP.NET 代码。

使用 ASP.NET 编写 Web 应用程序后，在客户端只需要一个浏览器即可预览执行程序。在 Web 服务器端需要安装 ASP.NET 运行库，即搭建 ASP.NET 处理的硬件环境平台。ASP.NET 服务器端必须具备的硬件环境如下。

- IIS：即虚拟服务器。
- .NET Framework：即.NET 框架。

16.2　创建一个简单的 ASP.NET 程序

在本节的内容中，将通过一个具体的实例来说明创建 ASP.NET 程序的过程。

实例 78：创建一个 ASP.NET 程序

本实例保存在文件夹“光盘：\daima\16”内，项目名称为 first。本实例的具体实现步骤如下。

1）打开 Visual Studio 2010，依次单击“文件”|“新建”|“网站”命令，弹出“新建网站”对话框，如图 16−2 所示。



图 16−2 “新建网站”对话框

2）设置位置为文件系统，语言为 Visual C#，命名为 first。单击“确定”按钮后将在指定目录上创建默认的 Web 程序，默认主页文件是 Default.aspx，如图 16-3 所示。

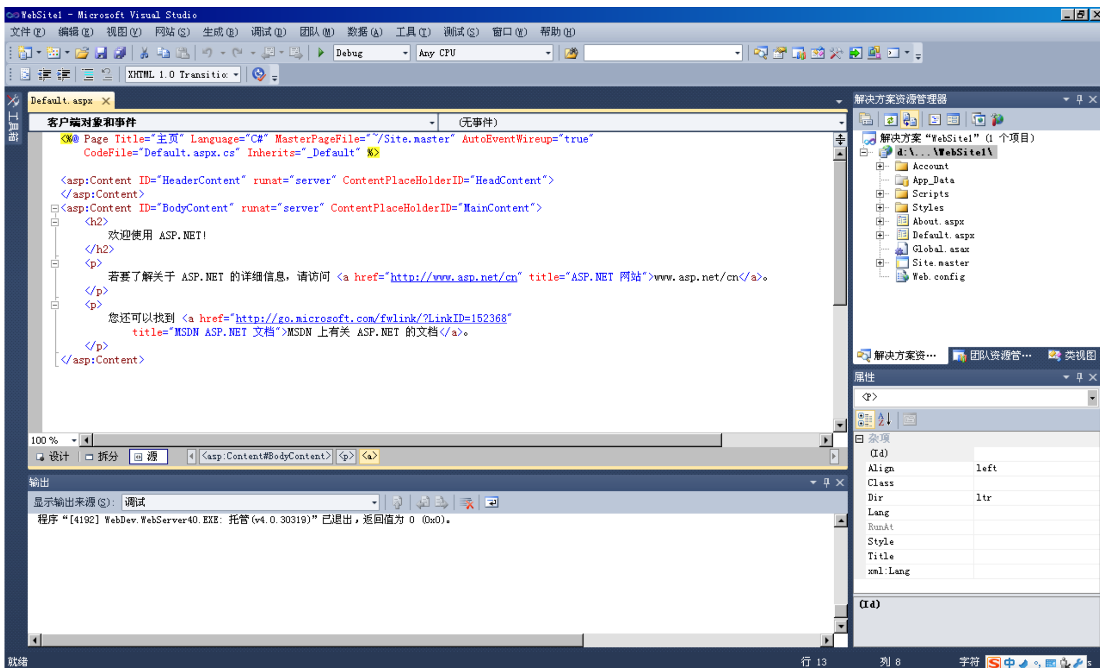


图 16-3　默认主页文件 Default.aspx

3）单击下方的“设计”标签来到主页的设计界面，如图 16-4 所示。

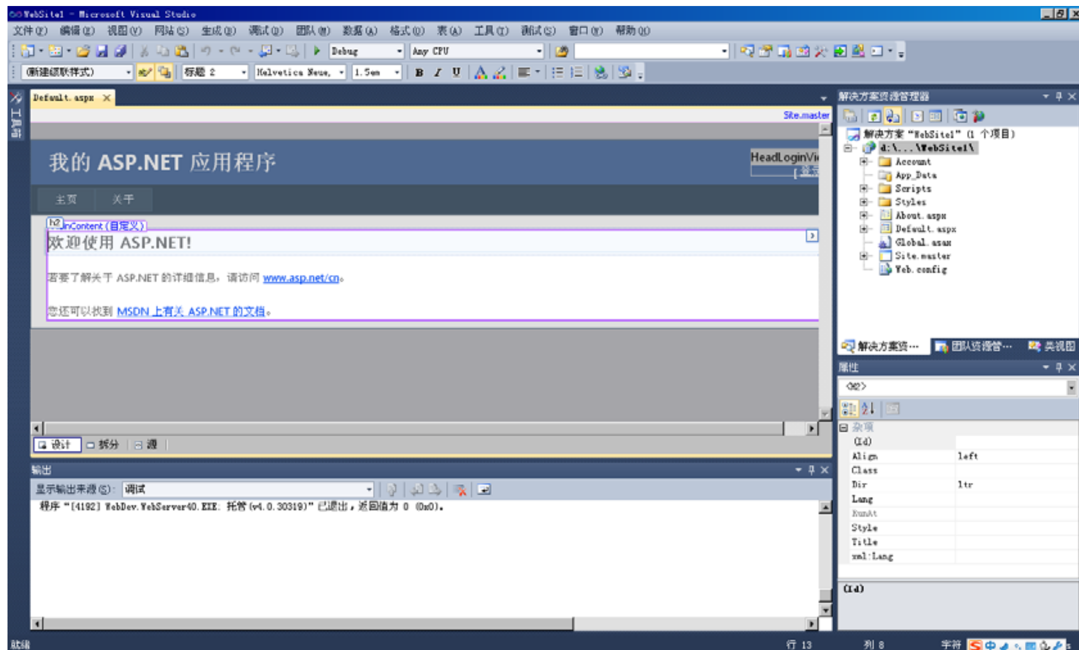


图 16-4　默认主页设计界面

4）单击“布局”｜“插入表”命令，在页面内插入一个 2×2 的表格。然后在表格内分别插入两个 Label 控件和 TextBox 控件，并分别设置他们的属性，如图 16-5 所示。

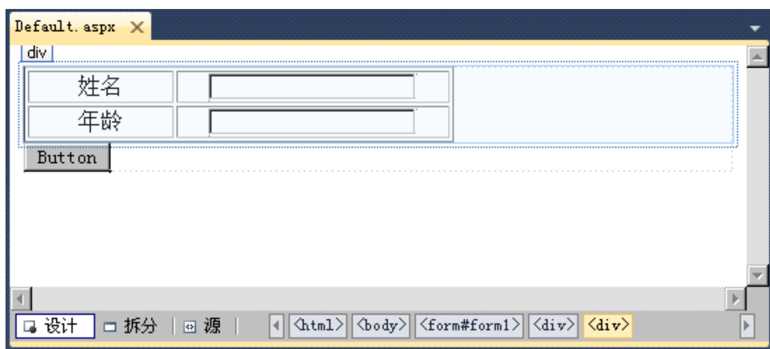


图 16-5　插入表格和控件

5）执行上面的主页后，会按照指定样式显示页面文件。单击下方的“源”标签来到主页的代码界面，查看其具体实现代码，具体代码如下：

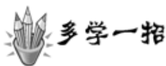
```
<%@Page Language="C#"AutoEventWireup="true" CodeFile="Default.aspx.cs"
Inherits="_Default” %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http:
//www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>无标题页</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>
        <div style="text-align: center">
            <table border="1">
                <tr>
                    <td style="width: 100px">
                        <asp:Label ID="Label1" runat="server" Text="姓名"
                        Width="92px"></asp:Label></td>
                    <td style="width: 191px">
                        <asp:TextBox ID="TextBox1" runat="server"></asp:
                        TextBox></td>
                </tr>
……………………………………………
    </form>
</body>
</html>
```

上述实例的代码都是自行生成的。其中文件 Default.aspx 的首行代码比较重要，各参数的具体说明如下。

- Language：设置页面代码的编写语言为 C#。
- AutoEventWireup：值为 True，设置页面内事件的处理程序自动连接到方法名上。
- CodeFile 和 Inherits：通过和 AutoEventWireup 相结合，将当前页面和对应的后台代码文件相关联，本实例页面的后台代码的关联文件是 Default.aspx.cs。

文件 Default.aspx.cs 是后台处理代码，在其中自动导入了一些命名空间，并且定义了类_Default，然后通过 Page_Load 将整个页面内容载入。文件 Default.aspx 的主要代码如下所示：

```
using System;
using System.Data;
using System.Configuration;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Web.UI.HtmlControls;
public partial class _Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
    }
}
```



通过 Visual Studio 使用 ASP.NET 可以轻松地编写 Web 应用程序，并且可以将后台代码文件 Default.aspx.cs 和前台显示文件 Default.aspx 独立保存，实现了处理和表现的完全分离。这样就使整个项目文件变得更加直观，便于日后对系统的维护。

16.3　ASP.NET 控件

在 ASP.NET 页面上的控件类型有如下三种。

1）HTML 服务器控件：能够模拟 HTML 元素，实现各页面元素的显示。

2）Web 服务器控件：其功能和 HTML 服务器控件类似，其属性和元素有一个公共命名模式，这样将便于程序开发，并且可以和相似的 Windows 窗体控件保持一致。常用的服务器控件可以分为标准控件和数据处理控件。

3）自定义控件：即为了满足特殊需求而自行编写的控件。

例如可以在 16.2 实例的主页中加入几个上述类型控件，因为原实例中已经使用了 Label 控件和 TextBox 控件，所以可以将插入的控件和事件联系起来，例如在 Default.aspx 中插入一个 Button 控件：

```
<asp:Button ID="Button1" runat="server" OnClick="Button1_Click" Text=
```

```
    "Button" />
```

然后鼠标单击事件 Click，则在后台文件 Default.aspx.cs 中会自动生成对应的事件处理文件：

```
    protected void Button1_Click(object sender, EventArgs e)
        {
        }
```

这样，Default.aspx 中的 OnClick 属性将告诉运行库在单击时会运行 Button1_Click 程序。

16.3.1 Web 服务器控件

几乎所有的 Web 服务器控件都继承了 System.Web.UI.WebControls.WebControl，而它又继承了 System.Web.UI.Control。Web 服务器控件有许多共同的属性和事件，通过本身的属性和事件不但能够控制控件的显示，而且能够进行相应的事件处理。

常用的 Web 服务器控件的具体信息如表 16−1 所示。

表 16−1 常用 Web 服务器控件信息

控 件	说 明
Lable	返回包含文本的 span 元素，返回的 HTML 格式是“<span>”
Literal	返回静态文本，返回的 HTML 格式是“static text”
TextBox	能够获取用户输入的信息，返回的 HTML 格式是“<input type=”text”>”
Button	将获取的信息传递给服务器，返回的 HTML 格式是“<input type=”submit”>”
LinkButton	创建一个超级链接，其中包含给服务器会送的 Javascript，返回的 HTML 格式是“<a href=javascript:dopostback()>”
ImageButton	创建一个图片标记，返回的 HTML 格式是“<input type=”image”>”
HyperLink	创建一个超级链接标记，返回的 HTML 格式是“<a>”
DropDownList	创建一个 select 标记，返回的 HTML 格式是“<select>”
ListBox	返回一个 check 类型的 input 标记，返回的 HTML 格式是“<select size=””>”
CheckBox	返回一个 check 类型的 input 标记，返回的 HTML 格式是“<input type=”checkbox”>”
RadioButton	返回一个 radio 类型的 input 标记，返回的 HTML 格式是“<input type=” radio”>”
Image	返回一个 radio 类型的 Image 标记，返回的 HTML 格式是“<Image>”
Calendar	可以显示完整的日历，并能输出带有 Javascript 代码的 HTML 表，返回的 HTML 格式是“<table>”
TreeView	返回一个 div 标记，里面包含了多个 table 标记，返回的 HTML 格式是“<div><table>”

实例 79：使用 Web 服务器控件

下面将通过一个简单的实例来说明 Web 服务器控件的具体使用过程。本实例保存在“光盘：\daima\16”文件夹内，项目名为 WebSite1，功能是在页面内实现动态数据输出显示的效果。本实例的实现流程如下。

1）在 Visual Studio 2010 中新建一个名为“WebSite1”的 ASP.NET 应用程序。

2）来到生成的默认文件 default.aspx，然后分别插入一个 DropDownList 控件、Button 控件和 Label 控件，如图 16−6 所示。

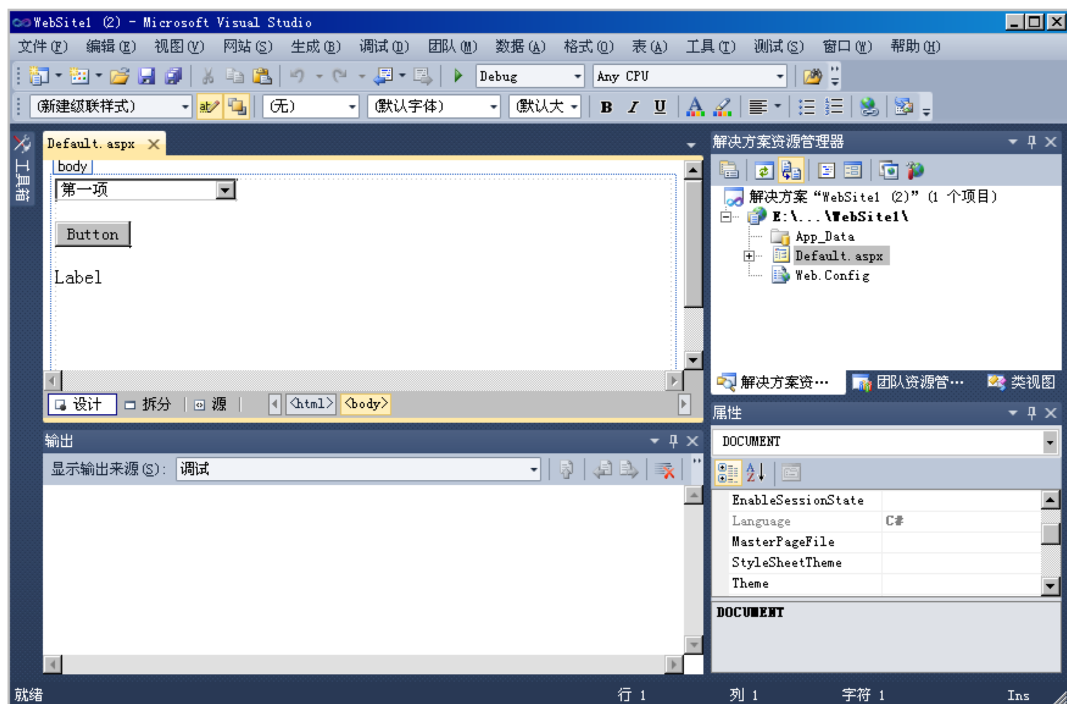


图 16-6　插入控件

3）选中 DropDownList，单击“属性”界面的 Items 后的图标...弹出“集合编辑器”对话框。

4）在对话框内为下拉框设置三个选项，如图 16-7 所示。

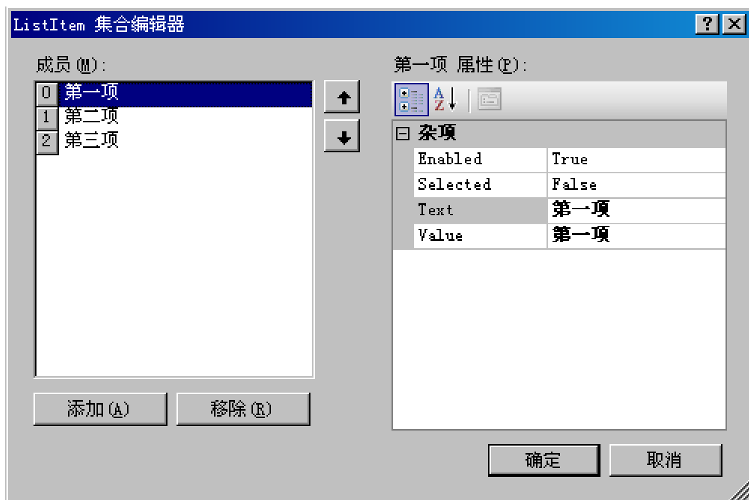


图 16-7　集合编辑器设置选项

5）双击 Button 按钮，来到自动生成的后台处理文件 default.aspx.cs 中。分别设置页面载入程序和按钮单击事件处理方法。具体代码如下：

```
public partial class _Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
    }
    protected void Button1_Click(object sender, EventArgs e)
    {
        this.Label1.Text="你选择的是："+this.DropDownList1.SelectedValue;
    }
    protected void DropDownList1_SelectedIndexChanged(object sender,
    EventArgs e)
    {
    }
}
```

页面执行后的显示效果如图 16-8 所示。当选中一个下拉选项并单击“Button”按钮后会执行单击事件的处理方法，显示对应的输出结果，如图 16-9 所示。

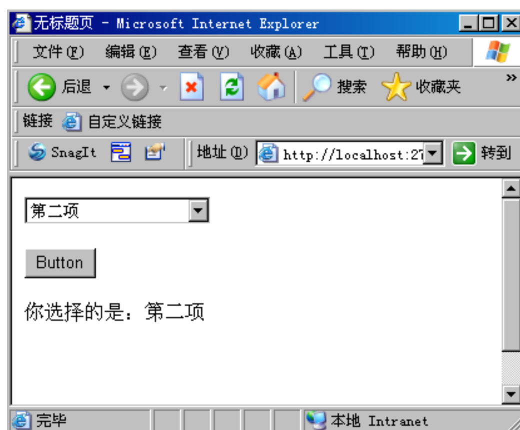


图 16-8　页面执行效果图

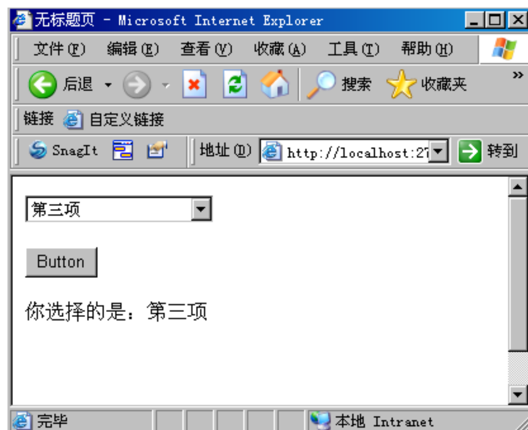


图 16-9　单击按钮后的效果

经过上述实例项目的设计流程可以看出，使用 Visual Studio 2010 不但可以轻松使用 ASP.NET 中的专用控件，而且可以大大减少代码编写数量。项目显示页面 default.aspx 的代码会自动生成，具体代码如下：

```
<% @ Page Language="C#" AutoEventWireup="true"  CodeFile="Default.
aspx.cs" Inherits="_Default” %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http:
//www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>无标题页</title>
</head>
<body>
    <form id="form1" runat="server">
```

```
    <div>
        <asp:DropDownList ID="DropDownList1" runat="server" Width=
      "149px" OnSelectedIndexChanged="DropDownList1_SelectedIndexChanged">
            <asp:ListItem>第一项</asp:ListItem>
            <asp:ListItem>第二项</asp:ListItem>
            <asp:ListItem>第三项</asp:ListItem>
        </asp:DropDownList><br />
        <br />
        <asp:Button ID="Button1" runat="server" OnClick="Button1_Click"
        Text="Button" /><br />
        <br />
        <asp:Label ID="Label1"runat="server"Height="105px"Text= "Label"
        Width="147px"></asp:Label> </div>
    </form>
</body>
</html>
```


多学一招

ASP.NET 的 Web 服务器控件是 ASP.NET 网页上的对象，在向浏览器请求页和呈现标记时将运行这些对象。许多 Web 服务器控件类似于熟悉的 HTML 元素，如按钮和文本框。其他控件具有复杂行为，如日历控件和管理数据连接的控件。

Web 服务器控件和 HTML 控件一样，在 ASP.NET 项目中也是被创建在服务器上的，并且也需要添加“runat="server"”。但是 Web 服务器控件比 HTML 服务器控件的书写格式要稍微复杂，需要加上固定的“asp:”标签。例如下面的代码:

```
<asp:Button ID="Button2" runat="server" OnClick="Button2_Click" Text=
"Button" />
<asp: Button>
```

16.3.2　数据处理控件

Web 数据处理控件是 Web 服务器控件的一种，能够实现和数据库数据的交互处理。Web 数据处理控件分为如下两类：

- 数据源控件：有 SqlDataSource、AccessDataSource、ObjectDataSource、XmlDataSource 和 SiteMapDataSource。
- 数据显示控件：有 GridView、DataList、DetailsView、FormView 和 Repeater。

在下面的内容中，将简要介绍上述两类控件的基本知识。

1．数据源控件

在通常情况下将数据源控件放在页面内，以建立和数据源的连接，这样就可以添加一个绑定控件来实现数据的显示。所有的数据源控件都派生于 System.Web.UI.DataSource 或 System.Web.UI.HierarchicalDataSource。通过上述类的方法，可以访问内部数据视图，并设置对应的显示样式。常用的 Web 数据源控件的具体信息如表 16-2 所示。

表 16-2　常用 Web 数据源控件信息

控　件	说　明
SqlDataSource	用于 SQL Server 数据库处理
AccessDataSource	用于 Access 数据库处理
ObjectDataSource	用于处理存储在自己创建的对象中的数据，这些对象可以组合在一个集合类中
XmlDataSource	和 DataSetDataSource 类似，但是能够处理层次结构化的数据
SiteMapDataSource	可以绑定到层次站点地图数据上

2．数据显示控件

数据显示控件能够实现数据库数据的显示，常用的 Web 数据显示控件的具体信息如表 16-3 所示。

表 16-3　常用 Web 数据显示控件信息

控　件	说　明
GridView	以数据行的格式显示多个数据项，其中每行对应着数据字段的列
DataList	显示多个数据项，可以为每一项提供模板，以任意指定方式显示数据字段。控件可以添加、编辑和删除数据项
DetailsView	以表格形式显示数据项，其中的每行和每列都和数据字段相对应。
FormView	使用模板显示一个数据项，可以添加、编辑和删除数据项。
Repeater	和 DataList 类似，但是不能选择和编辑数据

16.3.3　其他控件

除了上述服务器控件和数据处理控件外，在 Web 应用中还有其他几种常用控件。在下面的内容中，将简要介绍其他几种常用控件的知识。

1．验证控件

使用验证控件后，可以在不编写任何代码的前提下来验证用户输入的数据。除了上述功能外，验证控件还能在运行期间验证控件的有效性，并自动的为用户输出帮助提示。

验证控件可以生成客户端的 JavaScript 函数，这样可以简化验证任务的执行。所有的验证控件都基于 BaseValidator，它们通过共享的几个属性能够实现具体的功能。例如可以通过 ErrorMessage 属性可以验证控件所在的位置输出，即输出对应的验证结果。ControlToValidate 可以验证控件的编程 ID。

常用的 Web 验证控件的具体信息如表 16-4 所示。

表 16-4　常用 Web 验证控件信息

控　件	说　明
RequiredFieldValidator	用于验证在 TextBox 控件中输入的数据。
CompareValidator	用于验证用户输入的数据是否合法。
RangeValidator	用于验证控件中的数据，看其值是否在 Maximum 和 MinimumValue 属性之间的值 ，其 Type 属性对应每个 CompareValidato。
RegularExpressionValidator	根据存储在 ValidatorExpression 中正则表达式来验证字段的内容。
CustomValidator	使用定制的函数来验证控件中的数据。
ValidationSummary	为所有设置了 ErrorMessage 的验证控件显示验证错误。

实例 80：使用验证控件

下面将通过一个简单的实例来说明验证控件的具体使用过程。本实例保存在“光盘：\daima\16”文件夹内，项目名为 WebSite2，功能是对用户在表单内输入的数据进行验证处理。本实例的具体实现流程如下。

1）在 Visual Studio 2010 中新建一个名为“WebSite2”的 ASP.NET 应用程序。

2）来到生成的默认文件 default.aspx，然后分别插入两个 TextBox 控件，一个 Button 控件，三个 Label 控件，一个 CompareValidator 控件，如图 16-10 所示。

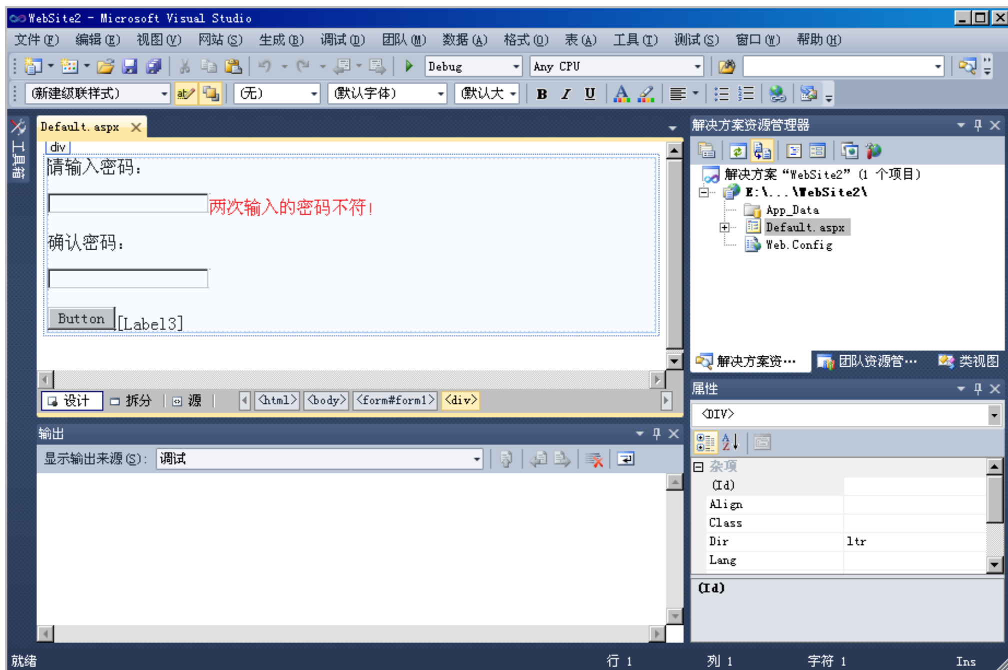


图 16-10 插入控件

3）选中 CompareValidator，设置其 ControlToCompare 属性值为“TextBox1”，设置其 ControlToValidator 属性值为“TextBox2”。

4）双击 Button 按钮，来到自动生成的后台处理文件 default.aspx.cs 中。分别设置页面载入程序和按钮单击事件处理方法。主要代码如下：

```
using System.Web.UI.HtmlControls;
public partial class _Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
    }
    protected void Button1_Click(object sender, EventArgs e)
    {
        Label3.Text = "密码正确！";
    }
}
```

执行后的显示效果如图 16-11 所示。如果输入的两个密码不同，当单击“Button”按钮后会显示对应的提示结果，如图 16-12 所示；如果输入的两个密码相同，当单击“Button”按钮后会显示对应的提示结果，如图 6-13 所示。

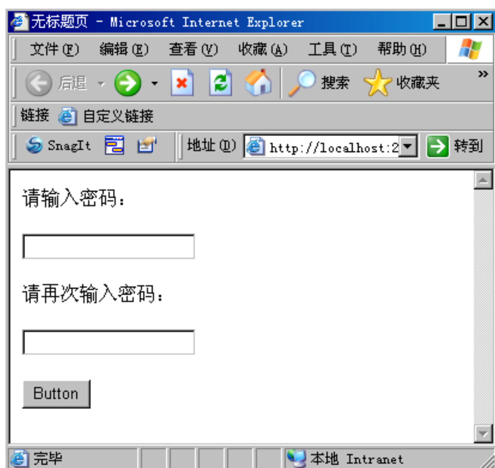


图 16-11　页面执行效果图



图 16-12　密码不同时的提示

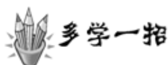


图 16-13　密码相同时的提示

项目设计完毕后，显示页面文件 default.aspx 的代码会自动生成，具体代码如下：

```
<% @ Page Language="C#" AutoEventWireup="true"  CodeFile="Default.aspx.
cs" Inherits="_Default” %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http:
//www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http:
//www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
<head id="Head1" runat="server">
    <title>无标题页</title>
```

```
    </head>
    <body>
        <form id="form1" runat="server">
        <div>
            <asp:Label ID="Label1" runat="server" Text="请输入密码："></asp:
            Label><br />
            <br />
            <asp:TextBox ID="TextBox1" runat="server" TextMode="Password">
            </asp:TextBox>
            <asp:CompareValidator ID="CompareValidator1" runat="server"
            ControlToCompare="TextBox1"
                ControlToValidate="TextBox2" ErrorMessage="两次输入的密码不符!
                "></asp:CompareValidator><br />
            <br />
            <asp:Label ID="Label2" runat="server" Text="确认密码："></asp:
            Label> <br />
            <br />
            <asp:TextBox ID="TextBox2" runat="server" TextMode="Password">
            </asp:TextBox><br />
            <br />
            <asp:Button ID="Button1" runat="server" OnClick="Button1_Click"
            Text="Button" />
            <asp:Label ID="Label3" runat="server"></asp:Label></div>
        </form>
    </body>
    </html>
```



在 Web 应用程序中，经常需要对用户的信息进行验证，确保合法时用户才能登录。在传统的 ASP 系统中，需要开发人员编写大量的代码来实现上述功能。但是在 ASP.NET 中，提供了专门的控件来实现验证处理。并且可以在服务器端对获取的客户端数据进行验证。在具体应用时，表 16-4 中的六种验证控件的方法基本一致。

2．导航和登录 Web 服务器控件

主要用于页面导航和用户的登录处理。

3．WebParts Web 服务器控件

通过 WebParts 提供的编辑功能，可以为用户提供全面的定制功能。

16.4 状态管理

HTTP 数据传输协议是没有状态的，当客户端和服务器间的连接请求结束后，连接也就随之关闭了。但是还需要将客户的信息从一个页面传到另一个页面。在 ASP.NET 中，可以通过如下方式存储当前的状态。

- ViewState。
- Cookie。
- 会话。
- 应用程序。
- 高速缓存。

16.4.1　ViewState

Web 服务器控件可以自动使用 ViewState 使事件工作，ViewState 包含的状态和控件发送给客户端时包含的状态相同。当浏览器把窗体发送给服务器时，ViewState 包含它的初始值，而发送控件中包含了新值。如果上面的初始值和新值有区别，就会调用相应的事件处理程序。

但是，如果在页面中使用了 ViewState，则会增加网络流量数据。这是因为数据需要从服务器端传递给客户端，再从客户端送到服务器端。可以禁止使用 ViewState，方法是在 Page 指令中把 EnableViewState 设置为 false：

```
<% @ Page Language="C#" AutoEventWireup="true"  CodeFile="Default.aspx.
cs" Inherits="_Default” EnableViewState=”false” %>
```

在一个控件内设置了它的 ViewState，就配置了该控件的 ViewState。并且可以把定制的数据存储在 ViewState，并使用索引符号来实现。这样就可以把保存在 ViewState 的数据进行网络传输，而不会影响它的状态，并且整个浏览器都支持这个特性，而不会造成因数据状态的失效而传送失败。

16.4.2　Cookies

在日常 Web 应用中，人们通常使用 Cookies 判断某个用户是否曾经进入本网站。具体来说，当用户第一次访问网站时，利用 Cookies 将数据存储到客户端计算机，当用户再次访问此网站时，则可利用 Request 对象的 Cookies 集合取得相关信息。

1．Cookies 的处理流程

Cookies 可以通过 HTTP Headers 从服务器端返回到浏览器上。首先，服务器端在响应中利用 Set-Cookies Headers 来创建一个 Cookies；然后，浏览器在它的请求中通过 Cookie Headers 包含这个已创建的 Cookies，并把它返回服务器，从而完成浏览器的认证。上述具体处理流程如图 16-14 所示。

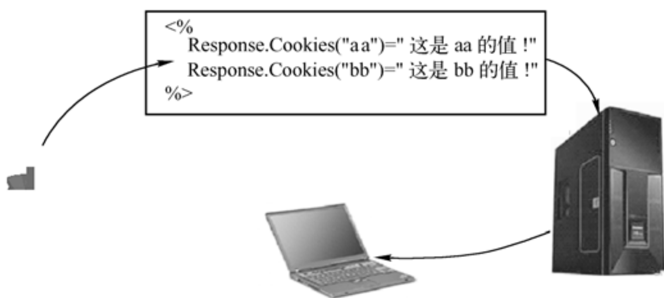


图 16-14　Cookies 处理流程图

2．Cookies 的特点

在 HTTP 协议下，Cookies 仅仅是一个文本文件，是服务器在浏览器端硬盘上写入的小文件。Cookies 存放在浏览器端的本机磁盘上，而不是存放在服务器端。通常，Cookies 包含用户的有关信息，如身份识别码和密码等。无论何时用户连接到服务器，Web 站点都可以访问 Cookies 信息。

使用 Cookies 的好处如下。

- ❑ 可以自由决定 Cookies 的生命周期，而不必担心 Cookies 会因自动消失而失去某些信息。例如，可以设置为数周、数月或数年。
- ❑ Cookies 存放在浏览器端的本机磁盘上，所以不会占用服务器端的磁盘空间。
- ❑ Cookies 可以记录浏览者的个人信息。这样站点制作者可以根据 cookie 记录的信息，设计出独具浏览者个人风格的网页或信息。

使用 Cookies 的缺点如下。

- ❑ 如果浏览者禁止服务器在浏览器端写入 Cookies，Cookies 就无法记录个人信息。
- ❑ 只能记录字符串、数值等简单的数据类型，无法记录对象、数组等复杂的数据类型。
- ❑ 因为 Cookies 被存放在浏览器端，所以可能被浏览者删除或拒绝写入。
- ❑ 会造成安全上的威胁，导致个人信息被窃取。

16.4.3 服务器端状态管理

前面介绍的 ViewState 和 Cookies 是客户端状态管理的技术，从下面开始将介绍几种服务器端状态管理的技术。

1．会话管理

会话的状态和浏览器的会话相关联，当服务器打开某页面时会话就已经开始了。如果用户在 20 分钟内没有访问服务器，则结束会话。

开发人员可以在 Global Application 类中编写自定义代码，设置会话开始或结束时的运行程序。Global Application 类的创建方法如下。

1）在 Visual Studio 中依次单击“网站”｜【添加新项】命令，在弹出的“添加新项”对话框中选择“全局应用程序类”命令，具体如图 16-15 所示。

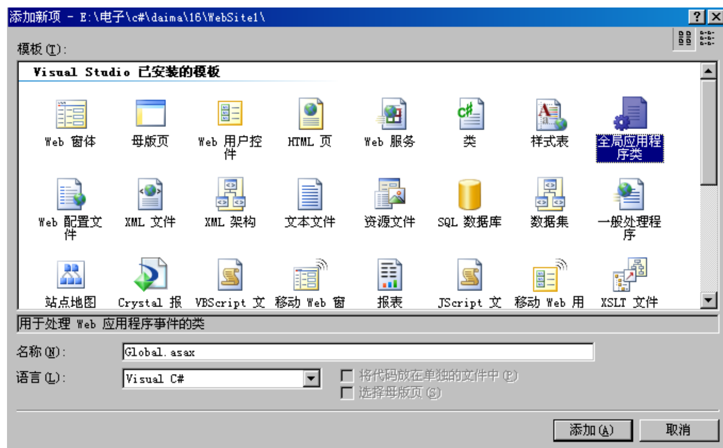


图 16-15 “添加新项”对话框

2）设置名称并单击“确定”按钮后即可创建完成。

在自动生成的文件内，定义了一些事件处理程序代码，具体如下：

```
<% @ Application Language="C#”% >
<script runat="server">
    void Application_Start(object sender, EventArgs e)
    {
        // 在应用程序启动时运行的代码
    }
    void Application_End(object sender, EventArgs e)
    {
        //  在应用程序关闭时运行的代码
    }
    void Application_Error(object sender, EventArgs e)
    {
        // 在出现未处理的错误时运行的代码
    }
    void Session_Start(object sender, EventArgs e)
    {
        // 在新会话启动时运行的代码
    }
    void Session_End(object sender, EventArgs e)
    {
        // 在会话结束时运行的代码。
        // 注意：只有在 Web.config 文件中的 sessionstate 模式设置为
        // InProc 时，才会引发 Session_End 事件。如果会话模式设置为 StateServer
        // 或 SQLServer，则不会引发该事件。
    }
</script>
```

这样可以将会话状态存储在 HttpSessionState 对象中，和当前 HTTP 环境相关的会话状态对象可以使用 Page 类的 Session 来访问。而在 Session_Start 处理程序中，可以初始化会话的变量。并且可以自定义处理事件，来完成会话状态的读取处理。

2．应用程序

如果数据需要在多个客户端共享，则可以使用应用程序状态。应用程序状态的具体使用方法和会话状态的非常类似，通过 Page 类的 Application 属性可以访问它。

看下面的代码：

```
void Application_Start(object sender, EventArgs e)
{
   Application[“Count”]=0;
}
```

经过编写上述代码后，一旦启动应用程序则会初始化应用程序变量 Count，而 Application_Start 是 global.asax 的事件处理方法。当在启动网站的第一个 ASP.NET 页面时，

就会立即调用 Application_Start，并且上述代码内的 Count 还能够统计网站的被访问数量。

3．高速缓存

高速缓存是服务器的状态，它类似于应用程序的运行状态。高速缓存比应用程序灵活，它可以通过多种方式定义某状态何时失效。存储后的数据直接被存储在高速缓存内。

16.5　安全性

为了保护整个网站信息的安全型，应该进行身份验证处理，检查当前用户是否有权登录。确保只有被授权的用户才能访问网站的资源。ASP.NET 使用如下三种身份验证方法。

1．Windows 身份验证

访问用户有个人的 Windows 账户，此验证方法一般在内联网站和局域网站点中使用。

2．Forms 身份验证

Web 站点可以维护它自己的用户列表，独立完成自己的身份验证。

3．Passport 身份验证

Passport 身份验证是微软提供的集中式身份验证服务。

现实中的大多数 Web 站点常用的是前两种方法。其中 Forms 身份验证是最通用的系统，能够迅速地建立安全的访问管理机制。实现 Forms 身份验证最简单的方法是使用 ASP.NET 配置工具，具体方法如下。

1）在 Visual Studio 2010 中依次单击“网站”|“ASP.NET 配置”命令后，即可进入安全配置向导，如图 16-16 所示。

2）在向导出现后，单击页面顶部导航栏上的“Security”图标，如图 16-17 所示。



图 16-16　开始 Forms 身份验证配置

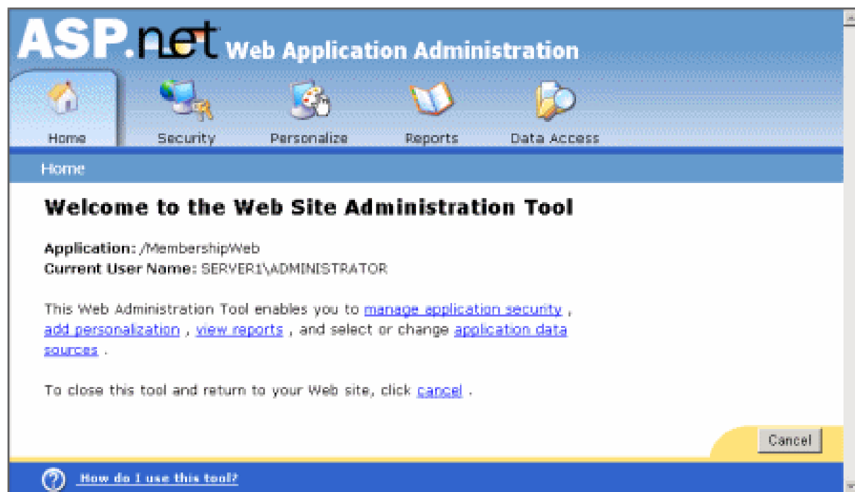


图 16-17　Web 应用程序管理工具

3）弹出的安全设置向导会提供两个选项：第一个选项在首次为 ASP.NET 站点设置安全时使用。第二个选项可以修改先前为站点配置的一组成员选项，如图 16-18 所示。

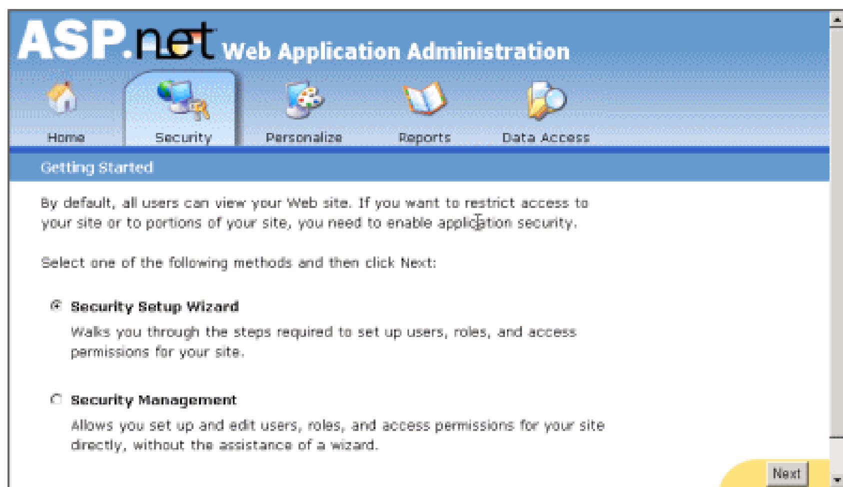


图 16-18　安全设置选项

4）跳过安全设置向导的欢迎界面后，将弹出访问方法设置界面，如图 16-19 所示。

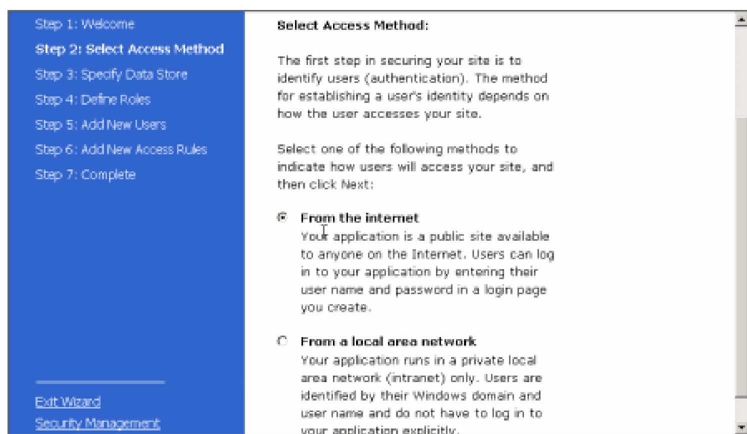


图 16-19　指定访问方法

在此可以设置用户是从 Internet 还是仅从本地的 Intranet 访问站点。需要注意的是，此处的选择将影响是否将站点设置为使用 Windows 身份验证和域登录，或者用户在进入站点时是否必须输入唯一的电子邮件和密码的组合。

如果想创建新数据库以跟踪站点的用户信息，则可在下一个屏幕中单击一个复选框。然后，需要指定用来存储成员信息的数据库的类型细节。可以选择在 Microsoft Access 或 Microsoft SQL Server 数据库中存储信息。

如果选择创建新数据库，则会提示指定数据库的名称和它的物理位置（对于 Access 数据库）或链接字符串信息（对于 SQL Server 数据库）。

下一步询问是否要启用某些安全设置。正如所看到的一样，无需编写代码，便可很容易

地为站点添加相当高级的身份验证机制。可以指定每个电子邮件（必须唯一），用户能够检索他们的密码，密码以安全格式存储以及用户能够在忘记密码时进行密码检索。还可以设置特定问题和相关答案，以便用户在检索密码时使用。正如所看到的一样，这些涵盖了密码管理的许多典型方面。

下一步让您指定站点的角色。虽然该步骤不是强制的，但最好使用基于角色的身份认证，因为它使您考虑应该允许给定的组执行的操作，而不是考虑允许特定用户执行的操作。然后可以轻松地把用户放置在这些角色中，并根据角色成员为其授予访问站点某些区域的权限。

添加角色只是指定想要添加的角色名称，然后在向站点添加用户时指定他们属于哪个角色。在添加用户时，可以指定他们的用户 ID、电子邮件地址、密码以及用于请求/响应的问题和答案（如果他们请求您将密码发送给他们）。如果您有所需要，该向导的另一部分还允许您向站点添加用户以及指定起始密码、密码请求问题和相关角色。

您还可以选择为站点指定身份验证或访问规则。这样，您可以为指定角色或用户限制对站点某些部分的访问。使用先前在向导中创建的用户和角色，您可以使用站点的树视图指定哪些用户和/或角色可以访问哪些部分。

通过完成上述向导后，可以转到安全管理页面并修改已添加的设置、创建新用户和角色以及删除用户和角色。

另外，文件 Web.config 是 ASP.NET 站点内的配置文件，可以在里面定义连接的数据库和定义需要的程序集。当新建一个 ASP.NET 程序后，如果在 VS05 种调试运行时会自动生成一个 Web.config 文件。具体代码如下：

```
<?xml version="1.0"?>
<configuration>
   <system.web>
        <compilation debug="true" targetFramework="4.0"/>
   </system.web>
</configuration>
```

在.NET 中，配置文件一般是按应用程序命名，如 myapp.exe.config。但是 web.config 比较特别，它必须放在应用程序虚拟目录的根目录，而且应用于该目录下所有网页和类。web.config XML 按纯文本方式存放，比较易于编辑和部署。

16.6　ASP.NET 和数据库的连接

ASP.NET 技术是一种 Web 编程技术，以满足系统的不同需求需要和不同的数据库工具进行连接。在本节的内容中，将向读者介绍 ASP.NET 和不同数据库工具的连接方法。

1．连接 Access 数据库

ASP.NET 建立和 Access 数据库的通用连接的代码格式如下：

```
using System.Web;
using System.Web.Security;
```

```
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Web.UI.HtmlControls;
using System.Data.Odbc;

public partial class _Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        //创建一个 ODBC 数据连接变量
        OdbcConnection oc = new OdbcConnection();
        //定义 ODBC 数据连接变量 oc 的连接字符串
        oc.ConnectionString = @"Driver={Microsoft Access Driver (*.
mdb)};dbq=D:\Personal\Desktop\Northwind.mdb;defaultdir=D:\Personal\Des
ktop;driverid=25;fil=MS Access;filedsn=D:\Personal\Desktop\access.dsn;
maxbuf fersize=2048;maxscanrows=12;pagetimeout=5;safetransactions=0;
threads=3;uid=admin;usercommitsync=Yes";
        try
        {
            oc.Open();  //连接
            Response.Write("<p>");
            Response.Write(oc.Database);
            Response.Write("数据库连接成功！");
            Response.Write("</p>");
        }
        catch (OdbcException oe)
        {
            Response.Write("<p>");
            Response.Write("数据库连接失败！");
            Response.Write(oe.Message);
            Response.Write("</p>");
        }
    }
}
```

2．连接 Oracle 数据库

ASP.NET 建立和 Oracle 数据库的通用连接的代码格式如下：

```
using System;
using System.Data;
using System.Configuration;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
```

```
    using System.Web.UI.HtmlControls;
    using Oracle.DataAccess.Client;
    public partial class _Default : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {
            //创建一个 OracleConnection 数据连接变量
            OracleConnection oc = new OracleConnection();
            //定义 OracleConnection 数据连接变量 oc 的连接字符串
            oc.ConnectionString = @"Data Source=myOracle;User ID=uid;pass
    word=pwd";
            try
            {
                Response.Write("<p>");
                oc.Open();  //连接
                Response.Write(oc.Database);
                Response.Write("数据库连接成功！");
                Response.Write("</p>");
            }
            catch (OracleException oe)
            {
                Response.Write("<p>");
                Response.Write("数据库连接失败！");
                Response.Write(oe.Message);
                Response.Write("</p>");
            }
        }
```

3．连接 MySql 数据库

ASP.NET 建立和 MySql 数据库的通用连接的代码格式如下：

```
    using System;
    using System.Data;
    using System.Configuration;
    using System.Web;
    using System.Web.Security;
    using System.Web.UI;
    using System.Web.UI.WebControls;
    using System.Web.UI.WebControls.WebParts;
    using System.Web.UI.HtmlControls;
    using MySql.Data.MySqlClient;
    public partial class _Default : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {
            //创建一个 MySqlConnection 数据连接变量
            MySqlConnection mc = new MySqlConnection();
```

```
            //定义 MySqlConnection 数据连接变量 oc 的连接字符串
            mc.ConnectionString = @"Server=localhost;User Id=root;Password=
            password; Persist Security Info=True;Database=mysql";
            try
            {
                Response.Write("<p>");
                mc.Open();  //连接
                Response.Write(mc.Database);
                Response.Write("数据库连接成功! ");
                Response.Write("</p>");
            }
            catch (MySqlException me)
            {
                Response.Write("<p>");
                Response.Write("数据库连接失败! ");
                Response.Write(me.Message);
                Response.Write("</p>");
            }
        }
    }
```

4．连接 OLE DB 数据库

ASP.NET 建立和 OLE DB 数据库的通用连接的代码格式如下：

```
    ……………………………………………
    using System.Data.OleDb;
    public partial class _Default : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {
            //创建一个 OleDb 数据连接变量
            OleDbConnection oc = new OleDbConnection();
            //定义 OleDb 数据连接变量 oc 的连接字符串
            oc.ConnectionString = @"Provider=Microsoft.ACE.OLEDB.12.0;Data
            Source=C:\ Northwind.mdb";
            try
            {
                Response.Write("<p>");
                oc.Open();  //连接
                Response.Write(oc.ToString());
                Response.Write("数据库连接成功! ");
                Response.Write("</p>");
            }
            catch (OleDbException oe)
            {
                Response.Write("<p>");
                Response.Write("数据库连接失败! ");
```

```
                Response.Write(oe.Message);
                Response.Write("<p>");
            }
        }
    }
```

16.7　疑难问题解析

本章详细介绍了 C#开发 Web 程序所需要的基本知识。本节将对本章中比较难以理解的问题进行讲解。

读者疑问：怎样使用高速缓存？

解答：要使用高速缓存则必须使用 System.Web.Caching 命名空间和 Cache 类，然后定义高速缓存对象来进行数据处理。高速缓存的具体使用流程如下。

1）使用 Page 类的 Cache 属性返回一个 Cache 对象。

2）使用 Cache 类的 Add()方法赋予高速缓存。其中 Add()方法的第一个参数定义了高速缓存项的名称，第二个参数定义了高速缓存的对象，第三个参数定义了缓存的依赖关系。

读者疑问：DropDownStyle 属性的各个取值是什么？

解答：在 ComboBox 控件内，DropDownStyle 属性可以确定要显示组合框的样式，其值是由 ComboBoxStyle 枚举定义的，具体说明如下。

- DropDown：文本可以编辑，用户必须单击箭头来显示列表。
- DropDownList：不能直接编辑文本，用户必须单击箭头来显示列表。只有在 AutoCompleteMode 为 Suggest 或 SuggestAppend 时才会显示列表。
- Simple：文本可以编辑，列表总是可见。



职场点拨——有错要勇于面对同事

长期和同事在一个公司内工作，既然有美好的事情发生，也就会有不愉快的事情发生。假如犯了一个错误，例如一时疏忽忘记实现一个功能，同事如果批评你，这时应该怎么办呢？如果总是觉得听到被别人指出错误是一件很没面子的事情，人家指出自己的错误，是一种耻辱，以下这些建议或能帮助克服这种心理障碍，慢慢懂得从批评中吸取教训。

1）要从根本上明白，别人的批评无损自己的价值，不要以敌视的态度对待意见与自己相左的同事。

2）如果别人对自己的工作表现有意见，或者传出对自己不满的风言风语，就说明自身肯定有不对的地方，而不是故意做对，要认真检讨自己，修补自己的错误。

3）不要把工作不被接受理解为自己不被接受。人无完人，每个人都会犯错误，即使开发经验自身，但是也有精神不足、工作过重和承受太沉重的生活压力的时候，这时会很容易犯错。如果在犯错后能以正确的态度面对它，错而能改，犯错便不是什么大事情，反而对于自己日后的工作和升职有很大好处。

第 17 章　数据库和 ADO.NET 操作

数据库的推出使编程语言步入了一个全新的领域，数据库技术是所有编程语言的核心。通过编程语言和数据库技术的结合，可以实现现实世界中所有和数据相关的处理系统。数据虽然可以存储在文件中，但是这样编程语言对文件的操作会比较麻烦，而且不利于对数据的维护和更新处理。如果通过专业的数据库工具存储后，不但可以方便地使用数据处理语言进行数据的管理和维护，而且能够保证数据的安全性。通过本章能学到如下知识。

- 什么是 ADO.NET。
- 安装 Northwind。
- 使用 DataReader。
- 命令和存储过程。
- DataSet 数据处理。
- ADO.NET 和 XML。
- 职场点拨——寻找更好的工作。

2008 年 XX 月 XX 日，阴

我感觉生活中总是存在很多不尽如人意的地方。上次跳槽的念头被 Wisdom 给浇灭了，现在也许是真的需要另谋高就了。



一问一答

小菜：我想换一个工作试试，你觉得呢？

Wisdom：既然决定了，我也不阻拦你了，但是我还是建议你要三思而后行

小菜：谢谢提醒！言归正传，一般编程项目中都需要数据库吧。

Wisdom：对，一般的项目都需要用到数据库。数据库是实现动态应用的最佳工具，数据库是当今大型应用程序开发必不可少的组成部分，通过数据库管理系统，可以有效地存储、管理和检索各种数据。就像你想跳槽改变自己的生活一样，数据库在一定程度上改变了软件产业的命运。

17.1　什么是 ADO.NET

ADO.NET 的主要功能，是以关系和面向对象的格式访问数据库。这里的数据库通常是常见的关系数据库，例如 Access 和 SQL Server2005 等。ADO.NET 也能够实现对非关系数据库的访问，并且可以集成到.NET Framework 中，从而可以使用任何语言进行数据访问。

ADO.NET 包括所有的 System.Data 命名空间以及它的嵌套命名空间，例如 System.Data.SqlClient 和 System.Data.OleDb 等，另外 System.Xml 命名空间中的和数据库相关的类也属于 ADO.NET。

17.1.1　数据库简介

ADO.NET 是为了解决数据库问题而推出的，所以说 ADO.NET 和数据库是不可分割的应用整体。在下面的内容中，将分别对 ADO.NET 的发展背景和数据库技术的基本知识进行简要说明。

1．数据库技术概述

随着计算机技术、通信技术和网络技术的飞速发展，信息系统渗透到社会各个领域。作为其核心的数据库技术更是得到了广泛的应用。数据的建设规模、数据库的信息量大小以及使用频度已成为衡量一个国家信息化程度的重要标志。数据库技术领域的常用概念如下。

（1）数据库管理（Database Administration）

数据库管理是有关建立、存储、修改和存取数据库中信息的技术，是指为保证数据库系统的正常运行和服务质量，有关人员须进行的技术管理工作。负责这些技术管理工作的个人或集体称为数据库管理员（DBA）。数据库管理的主要内容有：数据库的建立、数据库的调整、数据库的重组、数据库的重构、数据库的安全控制、数据的完整性控制和对用户提供技术支持。

（2）数据库

数据库是长期存储在计算机内有组织的大量共享的数据集合，它可以提供各种用户共享且具有最小冗余度和较高的数据与程序的独立性。

（3）数据模型

数据模型是现实世界数据特征的抽象，是数据技术的核心和基础。它是数据库系统的数学形式框架，是用来描述数据的一组概念和定义，主要包括如下方面的内容。

- ❑ 静态特征：对数据结构和关系的描述。
- ❑ 动态特征：在数据库上的操作，例如添加、删除和修改。
- ❑ 完整性约束：数据库中的数据必须满足的规则。

（4）概念模型

概念模型用于信息世界的建模，人们常常先将现实世界抽象为信息世界 ，然后将信息世界转换为机器世界。而概念模型是现实世界到机器世界的一个中间层次。

（5）数据模型

不同的数据模型具有不同的数据结构。目前最为常用的数据模型有层次模型、网状模型、关系模型和面向对象数据模型。其中层次模型和网状模型统称为非关系模型。

概念模型按照用户的观点对数据和信息进行建模，而数据模型按照计算机系统的观点对数据进行建模。

（6）关系数据模型

关系模型是当前应用最为广泛的一种模型。关系数据库都采用关系模型作为数据的组织方式。自从 20 世纪 80 年代以来，计算机厂商推出的数据库管理系统几乎都支持关系模型。

关系模型的基本要求是关系必须要规范，即要求关系模式必须满足一定的规范条件，关系的分量必须是一个不可再分的数据项。

（7）数据库系统的结构

设计数据库时，强调的是数据库结构；使用数据库时，关心的是数据库中的数据。从数据库系统角度看，数据库系统通常采用三级模式结构，这是数据库管理系统的内部系统结构。

（8）数据库管理系统

数据库管理系统即 DBMS，是数据库系统的核心，是为数据库的建立、使用和维护而配置的软件。它建立在操作系统的基础之上，是位于操作系统与用户之间的一层数据管理软件，负责对数据库进行统一的管理和控制。

数据库管理系统的功能主要包括如下六个方面：数据定义，数据操作，数据库运行管理，数据组织，存储和管理，数据库的建立和维护，数据通信接口。

2．ADO.NET 的产生背景

ADO.NET 来源于 COM 组件库 ADO，即 ActiveX Data Objects 的缩写，用于实现微软系统的数据库访问接口。通过现在的 ADO.NET 称谓后，微软希望它的数据访问方式将是.NET 编程的最优先方式。

ADO.NET 的功能和 ADO 类似，它能够提供常用的类集来实现对某类型数据库的访问，并且能够使用本身类、属性和方法，在.NET 这个大环境中发挥出更好的功效。

在原始的数据库数据访问技术中，不同类型的数据库数据是不相兼容的，每种类型的数据库都有自己独立的函数库。例如 Oracle 和 SQL Server 的就不相同。这就造成如果改变了数据库的类型，就应该相应地对整个系统的应用程序进行编写。为了解决上述问题，微软联合多个公司推出了 ODBC，开发人员通过 ODBC 后可以实现对任何数据库的访问，只要对系统程序的数据库连接代码修改后即可实现数据库平台的转换。

随着网络技术的继续发展和办公自动化的普及，其他类型的数据也需要进行相应的系统程序收集和处理。例如，电子邮件数据和 Office 数据等。此时 OLE DB 便解决上述问题，OLE DB 实现了数据库和访问程序间的抽象，实现了数据库和非数据库类型数据的处理。

而 ADO.NET 的推出原理和上面的 ODBC、OLE DB 类似，它能够同时支持对数据库数据和传统文件数据的访问。

17.1.2　ADO.NET 的目的

ADO.NET 开发模式的主要优点如下。

1．灵活而简捷地访问关系数据库

ADO.NET 被推出的主要目的是实现对关系数据库的访问，它能够使用本身的类比较容易地表示数据库中的表、列和行，并且使用其自身的 DataSet 类，可以明显地对数据库接口的访问功能进行扩展。

2．可扩展性

ADO.NET 不仅能够对数据库接口的访问功能进行扩展，并且为“插件.NET”的数据管理者提供了框架支持，上述管理者可以从任何数据源中读取数据。

ADO.NET 提供了几种常用的内置.NET 数据提供者，即数据库客户的命名空间。具体说

明如下。

- 用于 SQLServer 数据库。
- 用于 Oracle 数据库。
- 用于 ODBC。
- 用于 OLE DB。

从上述的内置数据的格式可以看出，ADO.NET 可以支持几乎所有的数据库或数据格式文件。

3．支持多层应用程序

ADO.NET 能够支持多层应用程序，实现了不同逻辑部分在不同层运行的功效，具体说明如下。

- 数据层：由数据库和实现数据库访问的代码组成。
- 业务层：事物具体处理层，包含业务逻辑。它定义了当前应用程序所实现的功能，并且能够将这个层和其他的层分离开来。
- 显示层：提供用户的处理界面，将系统元素显示在用户的面前。

4．对 XML 和关系数据库的访问进行了统一

ADO.NET 的重要目的之一是沟通行、列和 XML 文档中的关系数据。.NET 技术是以 XML 为基础构建的，ADO.NET 可以扩展.NET 的具体用法。

17.2　ADO.NET 的对象

ADO.NET 可以通过自身的类和对象，来实现对不同格式文件的数据处理。ADO.NET 类可以分为如下两种。

1）提供者对象：.NET Data 提供的对象，在其名称中包含这个提供者的名称。

2）用户对象：ADO.NET 的客户对象，是将数据读入到内存后用来访问和操作数据的对象。

17.2.1　提供者对象介绍

因为在提供者对象的名称中包含这个提供者的名称，所以用于 SQL Server.NET 提供者的类是 SqlConnection。在下面的内容中，将对常用的几种提供者对象进行简要说明。

1．连接对象

连接对象是数据库操作的基础，其功能是建立程序和数据源的连接。只有和数据源连接后，才能编写特定的程序代码实现对数据的处理。

2．命令对象

命令对象的功能是向被连接的数据库发出操作命令，例如数据的查询、删除和修改等。命令对象是数据库操作的核心，只有通过命令对象才能实现对数据库数据的更新操作处理。

3．CommandBuilder 对象

CommandBuilder 对象能够创建 SQL 命令，能够修改某查询表的数据。因为有不同的提供者，所以该对象的名称有 SqlCommandBuilder 和 OdbcCommandBuilder 等。

4．DataReader 对象

DataReader 对象能够读取数据源中的数据，但是只能读取向前的和只读的数据。和 CommandBuilder 对象相同，因为有不同的提供者，所以该对象的名称有 SqlDataReader 和 OdbcDataReader 等几种。

5．DataAdapter 对象

DataAdapter 对象是一个通用的类，可以对某数据源实现各种操作。同样因为有不同的提供者，所以该对象的名称有 SqlDataAdapter 和 OdbcDataAdapter 等几种。

17.2.2　用户对象介绍

在 System.Data 命名空间中，定义了几个 ADO.NET 的客户对象。因为它和数据的提供者无关，所以其名称是固定的。在下面的内容中，将对常用的几种用户对象进行简要说明。

1．DtaSet 对象

DtaSet 对象表示一组数据表，在应用程序中可以将上述数据表作为一个单元来引用。例如 student、class 和 teacher 是 DtaSet 中的一个表，分别表示学生、班级和老师。使用 DtaSet 对象可以迅速地获取上述表中的数据，并且在服务断开时可以实现对数据的更新处理。

2．DtaTable 对象

DtaTable 对象表示 DtaSet 内的一个表，DtaTable 可以通过 DtaColumn 对象和 DtaRow 对象来实现读取处理。其中 DtaColumn 表示表中的列，DtaRow 表示表中的行。

3．DtaRelation 对象

DtaRelation 对象表示了通过共享列而发生关系的两个表之间的关系，例如可以使用某表的编号 ID 来进行唯一性标识，并且通过此 ID 可以实现和其他表数据的某种联系对应。

17.2.3　System.Data 命名空间

在 C#中使用 ADO.NET 时，必须先引用 System.Data 命名空间，因为在 System.Data 命名空间中包含了所有的 ADO.NET 类。具体引用方法是将如下代码放在 ADO.NET 程序开端：

```
using System.Data;
```

经过上述引用后，就需要实现对数据源提供者的引用。因为有多种数据源提供者，所以需要根据具体的情况编写引用指令。

SQL Server.NET 的引用指令如下：

```
using System.Data.SqlClient;
```

Oracle.NET 的引用指令如下：

```
using System.Data.OracleClient;
```

OLE DB.NET 的引用指令如下：

```
using System.Data.OleDb;
```

ODBC.NET 的引用指令如下：

```
using System.Data.Odbc;
```



不开发人员应该根据具体项目的情况选择对应提供者的引用指令。例如，如果对移植性的要求比较高，则可以使用通用的 ODBC 方式；如果系统只是要求对某特定数据库的应用，则使用对应种类的.NET 数据库提供者。

17.3　安装 Northwind

Northwind 是微软公司提供的示例数据库文件，读者可以从微软的官方站点下载。本章的实例大多都是基于此数据库基础之上的，Northwind 的安装步骤和配置过程如下。

1）双击下载的安装文件“SQL 2000 SampleDb”，弹出安装对话框，如图 17-1 所示。

2）单击“Next”按钮，弹出“License Agreement”对话框，如图 17-2 所示。

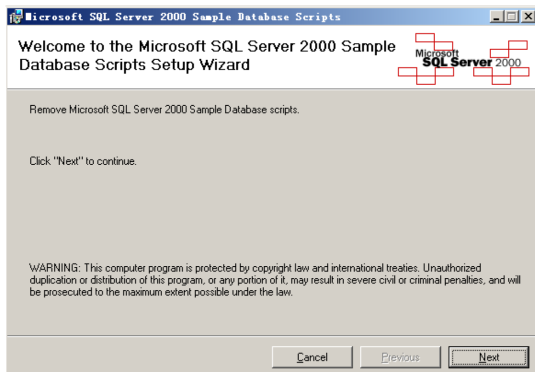


图 17-1　安装对话框

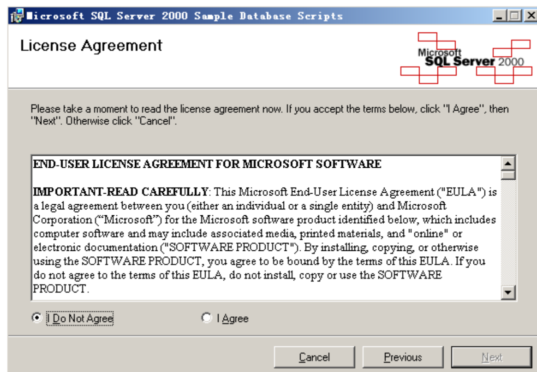


图 17-2　“License Agreement”对话框

3）选择“I Agree”选项，弹出安装对话框，如图 17-3 所示。

4）在弹出的“Choose Installation Options”对话框内单击“Next”按钮。如图 17-4 所示。

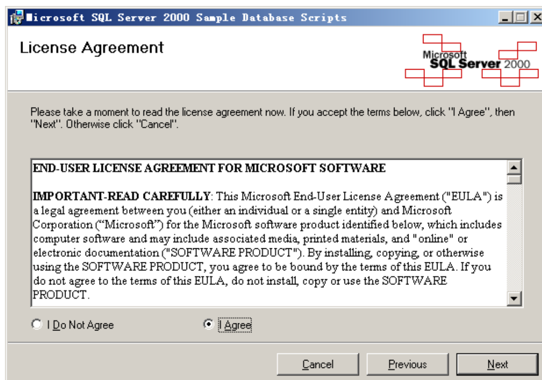


图 17-3　选择“I Agree”选项

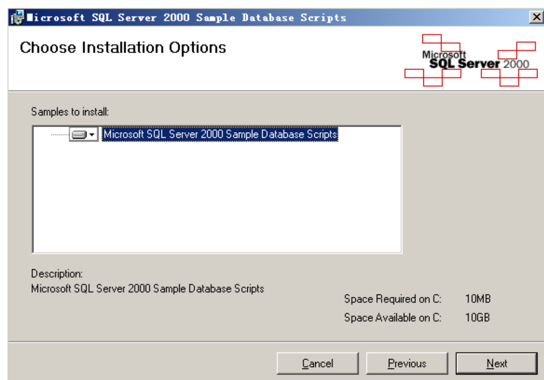


图 17-4　“Choose Installation Options”对话框

5）在弹出的“Confirm Installation 对话框内单击“Next”按钮，如图 17-5 所示。

6）开始进行安装，显示安装进度，如图 17-6 所示。

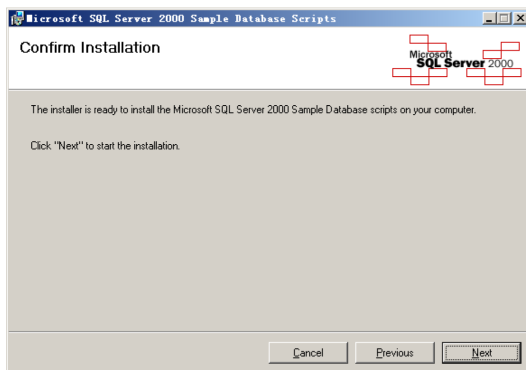


图 17-5　“Confirm Installation”对话框

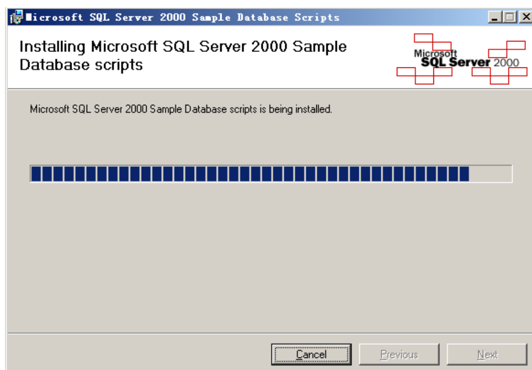


图 17-6　开始安装

7）安装完毕后单击“Close”按钮关闭安装对话框，如图 17-7 所示。

8）整个过程完成后，在默认的“C:\SQL Server 2000 Sample Databases”目录下将会保存 Northwind 的数据库示例，如图 17-8 所示。

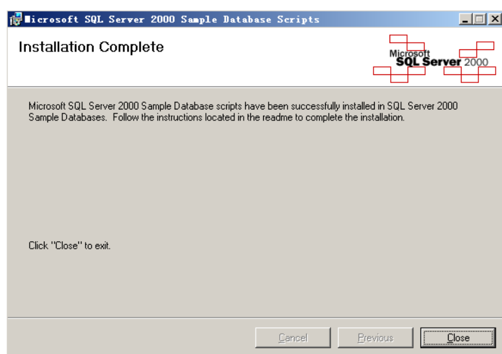


图 17-7　安装完成

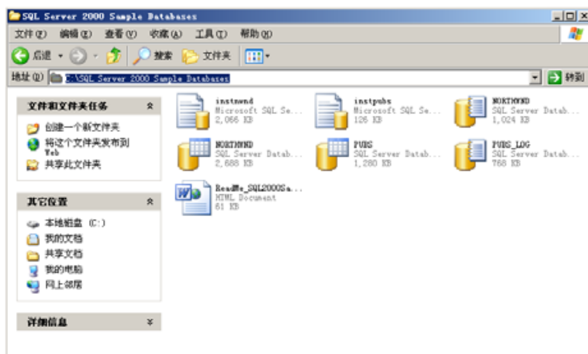


图 17-8　Northwind 的数据库示例

17.4　使用 DataReader

在本节的内容中，将通过一个具体的实例来说明 DataReader 的使用流程。

实例 81：使用 DataReader

本实例保存在“光盘:\daima\17”文件夹内，项目名为 DataReaderchuli。实例的功能是使用 DataReader 读取 Northwind 数据库中的数据。实例文件 DataReaderchuli.cs 的主要代码如下：

```
#region Using directives
using System;
using System.Data;
```

```
using System.Data.SqlClient;
#endregion
namespace DataReading
{
   class Program
   {
         static void Main(string[] args)
         {
               // Specify SQL Server-specific connection string
               SqlConnection conn = new SqlConnection(
                    @"Server=AAA\BBB;Integrated Security=True;" +
                    "Database=northwind");
               // Open connection
               conn.Open();
               // Create command for this connection
               SqlCommand thisCommand = conn.CreateCommand();
               // Specify SQL query for this command
               thisCommand.CommandText =
                     "SELECT CustomerID, CompanyName from Customers";
               // Execute DataReader for specified command
               SqlDataReader thisReader = thisCommand.ExecuteReader();
               // While there are rows to read
               while (thisReader.Read())
               {
                    Console.WriteLine("\t{0}\t{1}",
                    thisReader["CustomerID"], thisReader["CompanyName"
]);
               }
               // Close reader
               thisReader.Close();
               // Close connection
               conn.Close();
               Console.Write("执行完毕:");
               Console.ReadLine();
         }
   }
   }
```

上述实例文件的设计流程如下。

1）使用 using 指令引用 SQL Server 数据库。

2）定义对象 conn，设置和 Northwind 数据库的连接参数。

3）通过 SELECT 语句查询数据库内的数据。

4）通过对象 thisReader 读取并显示数据库的数据。

5）使用 Close 关闭 DataReader 和数据库连接。

执行后将在控制台内输出数据库的数据，如图 17-9 所示。

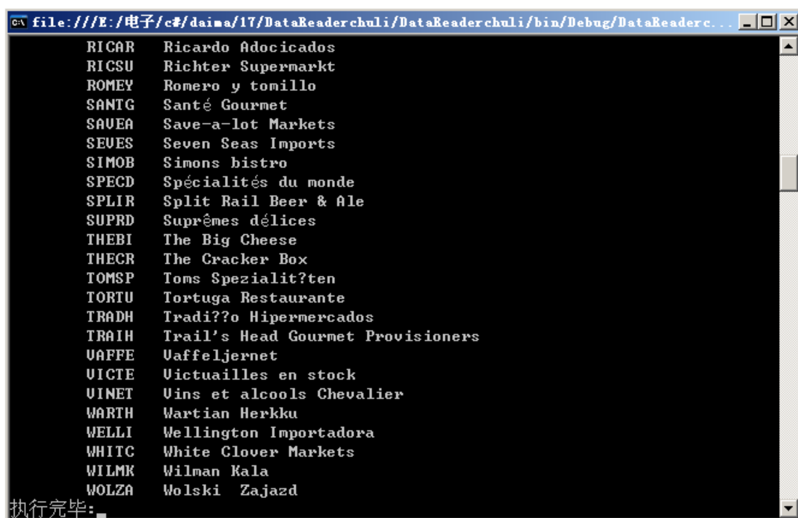


图 17-9　输出数据库数据


多学一招

DataReade 允许获得从 Command 对象的 SELECT 语句得到的结果。考虑性能的因素，从 DataReader 返回的数据都是快速的且只是“向前”的数据流。这意味着只能按照一定的顺序从数据流中取出数据。对于需要从数据库查询返回的结果中进行检索且一次处理一个记录的程序来说，它显得尤为重要。但是如果需要操作数据，更好的办法是使用 DataSet。DataReader 的主要属性信息如表 17-1 所示。

表 17-1　DataReader 属性信息

属　性	描　述
Depth	获取一个用于指示当前行的嵌套深度的值
FieldCount	获取当前行中的列数
HasRows	获取一个指示 SqlDataReader 是否包含一行或多行的值
IsClosed	检索一个布尔值，此值指示是否已关闭指定的 SqlDataReader 实例
Item	获取以本机格式表示的列的值
RecordsAffected	获取执行 Transact-SQL 语句所更改、插入或删除的行数
VisibleFieldCount	获取 SqlDataReader 中未隐藏的字段的数目
Connection	获取和 SqlDataReader 关联的 SqlConnection

17.5　数据库连接

为了实现应用程序对数据库的操作，需要首先建立和数据库的连接。在本节的内容中，将向读者介绍用 C#程序建立和数据库连接的方法。

17.5.1　基本连接处理

为了访问数据库，需要提供某种类型的连接参数，例如运行数据库的机器和登录凭证。以前用过 ADO 的用户，应该可以很快熟悉.NET 连接类 OleDbConnection 和 SqlConnection。使用下面的代码可以创建、打开和关闭 Northwind 数据库的连接：

```
    using System.Data.SqlClient;
    string source = "server= aaa\\bbb;" +
    "integrated security=SSPI;" +
    "database=Northwind";
SqlConnection conn = new SqlConnection(source);
conn.Open();
// Do something useful
conn.Close();
```

在上述代码的连接字符串中，各参数的具体说明如下。

- server= aaa\\bbb：表示要连接数据库服务器的名称，SQL Server 允许在同一台机器上运行多个不同的数据库服务器进程。
- integrated security=SSPI：这个参数使用 Windows Authentication 连接数据库，最好在源代码中使用这个参数，而不是用户名和密码。
- database=Northwind：描述了要连接的数据库实例。每个 SQL Server 进程都可以有几个数据库实例，这里设置的是使用 Northwind 实例。

SQL Server 还有另一个模式的身份验证，它可以使用 Windows 集成的安全性，这样在登录时提供的证书就会传送给 SQL Server。这样会适合于删除连接字符串的 uid 和 pwd 部分，并添加 Integrated Security=SSPI。

17.5.2　连接的使用

在一般情况下，当在.NET 中使用数据库连接时，要确保在使用每个资源后立即关闭。尽管.NET 的设计人员实现了自动的垃圾收集，垃圾最终都会被回收，但仍需要尽可能早地释放资源，以避免出现资源匮乏的情况。

当编写访问数据库的代码时，因为使连接打开的时间略长于需要的时间，这样可能会影响其他会话。在极端的情况下，不关闭连接会使其他用户无法进入一整组数据表，极大地降低了应用程序的性能。为此关闭数据库连接，要将一直处于打开状态的资源删除。

在 C#中，有如下两种方式可以确保数据库连接等类资源在使用后立即释放。

1．利用try…catch…finally 语句块

利用try…catch…finally 语句块实现连接优化的流程比较简单，例如下面的代码：

```
namespace ConnSqlDemo
{
    class Program
    {
        static void Main(string[] args)
```

```
            {
                //创建一个 SQL Server 数据连接变量
                SqlConnection mm = new SqlConnection();
                //定义 SQL Server 数据连接变量 mm 的连接字符串
                mm.ConnectionString = @"Data Source=AAA\BBB;Initial Catalog=
Northwind;Integrated Security=SSPI;";
                try
                {
                    mm.Open();  //连接
                    Console.WriteLine(mm.Database);  //输出数据库名称
                    Console.WriteLine("数据库连接成功！");
                }
                catch (SqlException nn)
                {
                    Console.WriteLine("数据库连接失败！");
                    Console.WriteLine(nn.Message);
                }
                Console.ReadLine();
            }
        }
    }
```

上述实例文件的设计流程如下。

1）使用 using 指令引用 SQL Server 数据库。

2）定义对象 mm，设置和 Northwind 数据库的连接参数。

3）利用try…catch…finally 语句块实现和 Northwind 数据库的连接。

将上述代码保存为“光盘:\daima\17\cachlianjie”，执行后将在控制台内输出数据库的数据，如图 17−10 所示。

Northwind
数据库连接成功!

图 17−10　输出数据库数据

在上述 finally 块中，可以释放已经使用的任何资源。这种方式的缺点是必须确保关闭连接，初学者很容易忘记在 finally 块中添加关闭连接的命令，所以应在编码风格上添加不容易出现反常情况的内容。

2．使用 using 语句块

在 C++中，只要某对象使用完毕后就会自动调用其析构函数。所以一旦出现了异常，若没有捕获，有析构函数的所有对象就会调用它们的析构函数。

在 C#和其他托管语言中，没有自动、决定性的析构方式，而是有一个垃圾收集器，它会在未来的某个时刻释放资源。这样并不能确定这个过程在什么时候发生，忘记关闭数据库连接可能会导致.NET 可执行程序的各种问题。但是可以使用 using 子句来解决问题，看下面一段代码：

```
string source = "server= AAA\BBB;" +
        " integrated security=SSPI;" +
        "database=Northwind";
using ( SqlConnection conn = new SqlConnection ( source ) )
{
```

```
// 打开连接
conn.Open ( ) ;
}
```

在上述代码中，通过 using 子句实现 IDisposable 接口对象在退出块时立即被释放。并且无论语句块是如何退出的，using 子句都会关闭数据库连接。

在编程时，至少应该使用上述两种方法中的一个，或者两种方法都使用。无论在哪里获得资源，最好都使用 using()语句。上述方式都没有好的异常处理方式来替代，所以在大多数情况下，最好组合使用这两种方法。例如下面的代码：

```
try
{
   using (SqlConnection conn = new SqlConnection ( source ))
   {
      conn.Open ( ) ;
      conn.Close ( ) ;
   }
}
catch (Exception e)
{
   …
}
```

这样，如果在 using 块中出现了异常，using 子句就会确保在资源上调用 IDisposable.Dispose 方法，在上述代码中将确保总是关闭数据库连接。

17.5.3　事务处理

在实际应用中需要对数据库要进行多次更新，这些更新必须在事务处理的范围内进行。ADO.NET 中的事务处理首先要在数据库连接对象上调用 BeginTransaction()方法，这些方法返回实现 IDbTransaction 接口的对象，而该对象是在 System.Data 中定义的。

例如，下面的代码序列是在 SQL Server 连接上开始事务处理的：

```
string source = "server= xxxxxx;" +
             " integrated security=SSPI;" +
             "database=Northwind";
SqlConnection conn = new SqlConnection(source);
conn.Open();
SqlTransaction tx = conn.BeginTransaction();
tx.Commit();
conn.Close();
```

在开始一个事务处理时，可以选择在该事务处理中执行的命令的优先级，这个优先级确定了如何在一个数据库会话中查看在另一个数据库会话中所进行的修改。常用命令的优先级如表 17−2 所示。

表 17-2　常用命令的优先级

优　先　级	说　　明
ReadCommitted	SQL Server 默认级别。这个级别可以确保只有第一个事务处理结束后，在第二个事务处理中才能访问第一个事务处理写入的数据
ReadUncommitted	即使一个事务处理还没有处理完数据，也允许另一个事务处理从数据库中读取数据。例如，如果两个用户在访问同一个数据库，第一个用户插入一些数据，但没有完成事务处理(通过 Commit()或 Rollback()方法)，第二个用户把它们的独立 级别设置为 ReadUncommitted，因此可以访问数据
RepeatableRead	这个级别扩展了 ReadCommitted 级别，如果在事务处理中使用了相同的语句，无论是否有其他潜在的数据库更新，总是可以返回相同的数据。这个级别要求对数据进行额外的锁定（这会降低性能），这个级别可以保证，对于初始查询的每一行，都不会修改数据，但允许显示“假想(phantom)”行—— 这些行是在进行事务处理时，由另一个事务处理插入的全新数据行
Serializable	这是最“高级”的事务处理级别，对数据库中的数据进行序列化的访问。利用这种独立级别，不会显示假想行，所以在序列化的事务处理中使用的 SQL 语句 总是检索相同的数据。序列化事务处理对性能的负面影响不应低估，如果肯定不需要这个独立级别，最好不要加上它

SQL Server 的默认优先级是 ReadCommitted，这是数据一致性和数据可用性的一种很好的折中，因为它比 RepeatableRead 或 Serializable 模式中需要的数据锁定都少。

17.6　命令和存储过程

通过处理命令和存储过程，可以实现对数据库数据的操作处理。在本节的内容中，将向读者介绍 C#处理命令和存储的基本知识。

17.6.1　处理命令

定义好命令并执行后就可以操作数据库中的数据。在类 Command 中提供了如下三个可执行的命令。

1）ExecuteNonQuery()：执行一个命令，但不返回任何结果。

2）ExecuteReader()：　执行一个命令，返回一个类型化的 IDataReader。

3）ExecuteScalar()：执行一个命令，返回一个值。

在类 SqlCommand 中，通过 ExecuteXmlReader()方法执行一个命令，返回一个 Xml Reader 对象，它可以用于传送从数据库中返回的 XML 代码段。

1．ExecuteNonQuery()

ExecuteNonQuery()一般用于执行 UPDATE、INSERT 和 DELETE 语句。例如在下面的代码中，使用 ExecuteNonQuery()指令对指定的数据进行了更新处理。

```
public class ExecuteNonQuerychuli
{
    public static void Main(string[] args)
    {
        string source = @"Data Source=xxxxx;Initial Catalog=Northwind;
Integrated Security=SSPI;";
        string select = "UPDATE Customers " +
                    "SET ContactName = '123' " +
                    "WHERE ContactName = 'mm'";
```

```
            SqlConnection conn = new SqlConnection(source);
            conn.Open();
            SqlCommand cmd = new SqlCommand(select, conn);
            int rowsReturned = cmd.ExecuteNonQuery();
            Console.WriteLine("修改的第{0}行。", rowsReturned);
            conn.Close();
            Console.ReadLine();
        }
    }
```

上述实例文件的设计流程如下。

1）使用 using 指令引用 SQL Server 数据库。

2）定义变量 source，设置和 Northwind 数据库的连接参数。

3）定义select 设置更新 SQL 语句。

4）关闭数据库连接，并返回被操作数据的行数。

将上述代码保存为“光盘:\daima\17\ExecuteNonQuery”，把库内 ContactName 值为 mm 的 ContactName 修改为 123，如图 17-11 所示。

图 17-11 返回被操作数据的行数

ExecuteNonQuery()返回命令语句所操作的行数，它是一个整数。

2．ExecuteReader()

ExecuteReader()能够根据使用的提供程序返回一个类型化的 DataReader 对象，返回的对象可以用于迭代返回的记录。例如在下面的代码中，使用 ExecuteReader()指令获取了指定数据库的数据。

```
    public class ExecuteNonQuerychuli
    {
        public static void Main(string[] args)
        {
            string source = @"Data Source=AAA\BBB;Initial Catalog=North
    wind;
    Integrated Security=SSPI;";
            string mm= "SELECT ContactName,CompanyName FROM Customers";
            SqlConnection conn = new SqlConnection(source);
            conn.Open();
            SqlCommand cmd = new SqlCommand(mm, conn);
            SqlDataReader nn = cmd .ExecuteReader();
            while (nn.Read())
            {
                Console.WriteLine("Contact : {0,-20} Company : {1}",
                                nn[0], nn[1]);
            }
            Console.ReadLine();
        }
    }
```

上述代码的设计流程如下。

1）使用 using 指令引用 SQL Server 数据库。

2）定义变量 source，设置和 Northwind 数据库的连接参数。

3）定义mm 设置更新 SQL 语句。

4）定义 SqlDataReader 对象 nn，并使用 ExecuteReader 指令执行 mm 对象的语句。

将上述代码保存为“光盘:\daima\17\ExecuteReader”，执行后将显示库内的指定数据，如图 17−12 所示。

```
Contact : Jytte Petersen       Company : Simons bistro
Contact : Dominique Perrier    Company : Spécialités du monde
Contact : Art Braunschweiger   Company : Split Rail Beer & Ale
Contact : Pascale Cartrain     Company : Suprêmes délices
Contact : Liz Nixon            Company : The Big Cheese
Contact : Liu Wong             Company : The Cracker Box
Contact : Karin Josephs        Company : Toms Spezialit?ten
Contact : Miguel Angel Paolino Company : Tortuga Restaurante
Contact : Anabela Domingues    Company : Tradi??o Hipermercados
Contact : Helvetius Nagy       Company : Trail's Head Gourmet Provisioners
Contact : Palle Ibsen          Company : Vaffeljernet
Contact : Mary Saveley         Company : Victuailles en stock
```

图 17−12　返回指定库的数据

3．ExecuteScalar ()

ExecuteScalar()可以从 SQL 语句返回一个结果，例如某表中的记录个数，或者服务器的当前日期或时间。例如在下面的代码中，通过 ExecuteScalar()指令返回了指定数据库表的行数。

```
public class ExecuteNonQuerychuli
{
   public static void Main(string[] args)
   {
       string source = @"Data Source=AAA\BBB;Initial Catalog=Northw
ind;
Integrated Security=SSPI;";
        string select = "SELECT COUNT(*) FROM Customers";
        SqlConnection conn = new SqlConnection(source);
        conn.Open();
        SqlCommand cmd = new SqlCommand(select, conn);
        object num = cmd.ExecuteScalar();
        Console.WriteLine(num);
        Console.ReadLine();
   }
}
```

上述实例文件的设计流程如下。

1）使用 using 指令引用 SQL Server 数据库。

2）定义变量 source，设置和 Northwind 数据库的连接参数。

3）定义mm 设置更新 SQL 语句。

4）定义 SqlDataReader 对象 nn，并使用 ExecuteReader 指令执行 mm 对象的语句。

将上述代码保存为“光盘:\daima\17\ExecuteScalar”，执行后将显示 Customers 表内数据的行数，如图 17-13 所示。



图 17-13　返回 Customers 表内数据的行数

4．ExecuteXmlReader()

ExecuteXmlReader()执行后将给调用者返回一个 XmlReader 对象。SQL Server 允许使用 FOR XML 子句来扩展 SQL 子句，上述子句可以是如下三个选项之一。

- ❑ FOR XML AUTO：根据 FROM 子句中的表建立一个树。
- ❑ FOR XML RAW：结果集中的行映射为元素，其中的列映射为属性。
- ❑ FOR XML EXPLICIT：必须指定要返回的 XML 树的形状。

ExecuteXmlReader 执行后将为调用者返回一个 XmlReader 对象。SQL Server 允许使用 FOR XML 子句来扩展 SQL 子句，上述子句可以是如下三个选项之一。

- ❑ FOR XML AUTO：根据 FROM 子句中的表建立一个树。
- ❑ FOR XML RAW：结果集中的行映射为元素，其中的列映射为属性。
- ❑ FOR XML EXPLICIT：必须指定要返回的 XML 树的形状。

例如在下面的代码中，通过 ExecuteXmlReader()指令将数据库内的数据建立了一个树：

```
public class ExecuteNonQuerychuli
{
    public static void Main(string[] args)
    {
         string source = @"Data Source=AAA\BBB;Initial Catalog=Northw
ind;
Integrated Security=SSPI;";
         string select = "SELECT ContactName,CompanyName " +
                  "FROM Customers FOR XML AUTO";
      SqlConnection conn = new SqlConnection(source);
      conn.Open();
      SqlCommand cmd = new SqlCommand(select, conn);
      XmlReader xx= cmd.ExecuteXmlReader();
      xx.Read();
string mm;
      do
      {
         mm= xx. ReadOuterXml();
         if (mm!="")
            Console.WriteLine(mm);
      }
      while (mm!="");
      conn.Close();
        Console.ReadLine();
    }
```

```
        }
```

将上述代码保存为“光盘:\daima\17\ExecuteXmlReader”，执行后将输出显示创建的树结构代码，如图 17-14 所示。

```
<Customers ContactName="Art Braunschweiger" CompanyName="Split Rail Beer &amp; A
le" />
<Customers ContactName="Pascale Cartrain" CompanyName="Suprêmes délices" />
<Customers ContactName="Liz Nixon" CompanyName="The Big Cheese" />
<Customers ContactName="Liu Wong" CompanyName="The Cracker Box" />
<Customers ContactName="Karin Josephs" CompanyName="Toms Spezialit?ten" />
<Customers ContactName="Miguel Angel Paolino" CompanyName="Tortuga Restaurante"
/>
<Customers ContactName="Anabela Domingues" CompanyName="Tradi??o Hipermercados"
/>
<Customers ContactName="Helvetius Nagy" CompanyName="Trail's Head Gourmet Provis
ioners" />
<Customers ContactName="Palle Ibsen" CompanyName="Vaffeljernet" />
<Customers ContactName="Mary Saveley" CompanyName="Victuailles en stock" />
<Customers ContactName="Paul Henriot" CompanyName="Vins et alcools Chevalier" />
```

图 17-14　返回树结构代码



必须导入 System.Xml 命名空间后才能输出返回的 XML。本例在 SQL 语句中包含了 FOR XML AUTO 子句，然后调用 ExecuteXmlReader()方法。

在 SQL 子句中，指定了 FROM Customers，这样类型 Customers 的元素就显示在输出中。为它添加元素，每个元素对应于从数据库中选择出来的列。

17.6.2　存储过程

当使用一个命令对象调用存储过程后，就实现了对存储过程的名称定义。通过定义存储过程，可以实现数据库的多种操作。下面以数据库 Northwind 内的 Region 为例，说明存储过程的处理。

1．调用没有返回值的存储过程

调用存储过程的最简单示例是不给调用者任何返回值。下面定义了两个这样的存储过程，一个用于更新 Region 记录，另一个用于删除指定的 Region 记录。

（1）记录的更新

因为只能更新一个列，所以可以比较很简单地更新 Region 的记录操作。只需直接在 SQL Server 查询分析器中输入对应代码，即可实现存储过程处理。看下面的存储过程：

```
CREATE PROCEDURE Regionxiugai (@RegionID INTEGER,
                              @RegionDescription NCHAR(50)) AS
  SET NOCOUNT OFF
  UPDATE Region
    SET RegionDescription = @RegionDescription
    WHERE RegionID = @RegionID
GO
```

如果为数据库表执行更新命令，需要重复选择和返回已更新的记录。这个存储过程有两个输入参数@RegionID 和@RegionDescription，对数据库执行 UPDATE 语句。

要在.NET 代码中运行这个存储过程，需要定义一个 SQL 命令并执行这个指令，例如下面的代码：

```
SqlCommand mm = new SqlCommand("Regionxiugai", conn);
mm.CommandType = CommandType.StoredProcedure;
mm.Parameters.Add(new SqlParameter ("@RegionID",
              SqlDbType.Int,
              0,
              "RegionID"));
mm.Parameters.Add(new SqlParameter("@RegionDescription",
              SqlDbType.NChar,
              50,
              "RegionDescription"));
mm.UpdatedRowSource = UpdateRowSource.None;
```

在上述代码中，创建了一个新 SqlCommand 对象 mm，并把它定义为一个存储过程。然后，依次添加每个参数，最后把存储过程的结果设置为 UpdateRowSource 枚举中的一个值。

该存储过程有如下两个参数。

- ❑ 要更新的 Region 记录的唯一主键码。
- ❑ 给这个记录的新描述。

创建好执行命令后，就可以用下面的命令来执行：

```
mm.Parameters[0].Value = 999;
mm.Parameters[1].Value = "South Western England";
mm.ExecuteNonQuery();
```

上述命令设置了每个参数的值，然后执行存储过程。命令参数可以像前面那样依次设置，或者按照名称来设置。

（2）记录的删除

通过下面的存储过程，可以从数据库中删除一个 Region 记录。

```
CREATE PROCEDURE RegionDelete (@RegionID INTEGER) AS
  SET NOCOUNT OFF
  DELETE FROM Region
  WHERE RegionID = @RegionID
GO
```

上述过程只需要该记录的主键码值。

在下面的代码中，通过 SqlCommand 对象 mm 调用了上述存储过程。

```
SqlCommand mm = new SqlCommand("Regionshanchu " , conn);
mm.CommandType = CommandType.StoredProcedure;
mm.Parameters.Add(new SqlParameter("@RegionID" , SqlDbType.Int , 0 ,
```

```
                                            "RegionID"));
    mm.UpdatedRowSource = UpdateRowSource.None;
```

上述命令只接收一个参数。

通过如下代码可以执行 Regionshanchu 存储过程：

```
    mm.Parameters["@RegionID"].Value= 999;
    mm.ExecuteNonQuery();
```

2．调用返回输出参数的存储过程

上面的两个存储过程的执行结果都没有返回值。如果存储过程包含输出参数，则这些参数需要在.NET 客户程序中定义，以便在过程返回时填充其输出参数。

Region 表仅由一个主键码（RegionID）和描述字段（RegionDescription）组成。要插入一个记录，需要生成该数字主键码，再把新行插入到数据库中。这样可以通过在存储过程中创建一个主键码来简化主键码的生成。具体代码如下：

```
    CREATE PROCEDURE RegionInsertchar(@RegionDescription NCHAR(50),
                  @RegionID INTEGER OUTPUT)AS
      SET NOCOUNT OFF
      SELECT @RegionID = MAX(RegionID)+ 1
      FROM Region
      INSERT INTO Region(RegionID, RegionDescription)
      VALUES(@RegionID, @RegionDescription)
    GO
```

通过上述插入过程创建了一个新的 Region 记录，在数据库本身生成主键码值时，这个值作为输出参数从过程返回（@RegionID）。如果是比较复杂的表就通常不会使用输出参数，而是将整个行插入，把该行返回给调用者。看下面的代码：

```
    SqlCommand  mm = new SqlCommand("RegionInsertchar" , conn);
    mm.CommandType = CommandType.StoredProcedure;
    mm.Parameters.Add(new SqlParameter("@RegionDescription" ,
                 SqlDbType.NChar ,
                 50 ,
                 "RegionDescription"));
    mm.Parameters.Add(new SqlParameter("@RegionID" ,
                 SqlDbType.Int,
                 0 ,
                 ParameterDirection.Output ,
                 false ,
                 0 ,
                 0 ,
                 "RegionID" ,
                 DataRowVersion.Default ,
                 null));
```

```
    mm.UpdatedRowSource = UpdateRowSource.OutputParameters;
```

第二个参数@RegionID 定义为包含其参数定向，在这个示例中是 Output。并且在最后一行使用了 UpdateRowSource 枚举来表示通过输出参数从这个存储过程返回的数据。当从一个 DataTable 中执行存储过程调用时，一般使用这个标志。

调用上述存储过程类似于前面的存储过程调用，但在上述代码中需要在执行完过程后读取输出参数，调用代码如下：

```
    mm.Parameters["@RegionDescription"].Value = "South West";
    mm.ExecuteNonQuery();
    int newRegionID = (int) mm.Parameters["@RegionID"].Value;
```

在执行完命令后，读取@RegionID 参数的值，并把它的数据类型转换为整型。

如果调用的存储过程返回输出参数和一组记录行，则可以在上述代码中定义合适的参数，而不需要调用 ExecuteNonQuery()，而应该调用 ExecuteReader()。

17.7　DataSet 数据处理

DataSet 是 ADO.NET 的核心对象之一，所有的复杂数据库操作一般都会需要用到它。在 DataSet 中包含了一组 DataTable 对象，表示被操作数据库的表。DataTable 有 DataRow 和 DataColumn 两个子对象，分别用来表示数据表的行和列。通过 DataRow 和 DataColumn 子对象，可以获取数据库的表、行和列的数据。

17.7.1　数据表

数据表非常类似于物理数据库表，由一些带有特定属性的列组成。数据表也可以定义主键码，在列上可以包含对应的约束。DataTable 对象可以附带任意多个扩展属性，可以用附属于对象的用户自定义信息来填充这个属性。当数据在中间层中构造，要返回给客户机，进行某些处理时，最适合使用扩展的属性。例如，可以在扩展的属性(如 min 和 max)中存储数字列的有效性标准，在验证用户输入时在 UI 层使用它们。

填充数据表时，可以从数据库中选择数据，从文件中读取数据，或在代码中手工填充，Rows 集合会包含这些检索出来的数据。

Columns 集合包含已经添加到表中的 DataColumn 实例，它们定义了数据的模式，例如数据类型、是否可为空和默认值等。

17.7.2　数据列

DataColumn 对象定义了数据表中某列的属性，既可以在代码中创建这个列，也可以由运行库自动生成。可以在构造函数中提供列的数据类型，也可以通过设置 DataType 属性来指定。把数据加载到数据表中后，就不能改变列的数据类型了，否则会抛出 ArgumentException 异常。

可以使用表 17-3 中的.NET Framework 数据类型来创建数据列。

表 17-3 数据列可以使用的.NET Framework 数据类型

Boolean	Decimal	Int64	TimeSpan
Byte	Double	Sbyte	UInt16
Char	Int16	Single	UInt32
DateTime	Int32	String	UInt64

可以给 DataColumn 设置的属性种类如表 17-4 所示。

表 17-4 可设置的 DataColumn 属性

属　性	说　明
AllowDBNull	如果为 true，该列就可以设置为 DBNull
AutoIncrement	定义自动生成的列值为一个递增的数字
AutoIncrementSeed	定义 AutoIncrement 列最初的种子值
AutoIncrementStep	用默认的步骤定义自动生成列值的步骤
Caption	可以用于在屏幕上显示列名
ColumnMapping	指定当 DataSet 通过调用 DataSet.WriteXml 来保存时，列如何映射到 XML 上
ColumnName	列名，如果没有在构造函数中设置，就由运行库自动生成
DataType	列的 System.Type 值
DefaultValue	可以定义列的默认值
Expression	定义表达式用于所计算的列

17.7.3 数据行

可以根据 DataTable 类来定义数据表中的列，而表中的数据是用 DataRow 对象来访问的。

实例 82：使用 DataRow 对象

下面通过一个具体的实例来说明使用 DataRow 对象的过程。本实例代码保存在“光盘:\daima\17”文件夹内，项目名为 DataRowchuli。实例的功能是将数据库数据建立一个树。实例文件 DataRowchuli.cs 的主要代码如下：

```
using System;
using System.Collections.Generic;
using System.Data.SqlClient;
using System.Data;
using System.Text;
namespace DataRowchuli
{
    class Program
    {
        static void Main(string[] args)
        {
            string source = "server=GUAN\\AAA;" +
             "uid=sa;pwd=888888;" +
             "database=northwind";
            string select = "SELECT ContactName,CompanyName FROM Custome
rs";
            SqlConnection conn = new SqlConnection(source);
```

```
                SqlDataAdapter mm = new SqlDataAdapter(select, conn);
                DataSet nn = new DataSet();
                mm.Fill(nn, "Customers");
                foreach (DataRow row in nn.Tables["Customers"].Rows)
                    Console.WriteLine("'数据{0}'和{1}同行", row[0], row[1]);
                conn.Close();
                Console.ReadLine();
            }
        }
    }
```

上述实例文件的实现流程如下。

1）定义连接参数 source，实现和指定数据库 northwind 的连接。

2）定义查询语句参数 select 查询库内 Customer 表的信息。

3）定义 SqlDataAdapter 类对象 mm，用于选择 DataSet 中的数据。

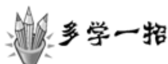
4）定义 DataSet 对象 nn，使用 Fill()方法对表的数据进行处理。

5）使用 DataRow 的索引器访问各数据行上的值。指定列的值可用几个重载的索引器来检索，这样就可以通过已知的列号、列名或 DataColum 来检索数据的值。

执行后将输出显示各对应同行元素的值，如图 17-15 所示。

```
数据Maria Anders'和Alfreds Futterkiste同行
数据Ana Trujillo'和Ana Trujillo Emparedados y helados同行
数据Antonio Moreno'和Antonio Moreno Taquería同行
数据Thomas Hardy'和Around the Horn同行
数据Christina Berglund'和Berglunds snabbk?p同行
数据Hanna Moos'和Blauer See Delikatessen同行
数据Frédérique Citeaux'和Blondesddsl père et fils同行
数据Martín Sommer'和Bólido Comidas preparadas同行
数据Laurence Lebihan'和Bon app'同行
数据Elizabeth Lincoln'和Bottom-Dollar Markets同行
数据Victoria Ashworth'和B's Beverages同行
数据Patricio Simpson'和Cactus Comidas para llevar同行
数据Francisco Chang'和Centro comercial Moctezuma同行
数据Yang Wang'和Chop-suey Chinese同行
```

图 17-15　输出显示各对应同行元素的值



多学一招

在使用过程中，DataRow 可以接收某一行上指定列的各个值，各版本（Version）的具体说明如表 17-5 所示。

表 17-5　DataRow 各取值说明

DataRow 的版本值	说　　明
Current	列中目前存在的值，如果没有进行编辑，该值与初值相同。如果进行了编辑，该值就是最后输入的一个有效值
Default	默认值
Original	最初从数据库中选择出来的列值，如果调用了 DataRow 的 AcceptChanges()方法，该值就更新为当前值
Proposed	对列进行逐步的修改时，可以检索到这个已改变的值。如果在行上调用了方法 BeginEdit()，并进行了修改，每一列都会有一个推荐值，直到调用了 EndEdit()或 CancelEdit()为止

如果从 DataRow 索引器中检索某个版本的值，应使用索引器方法，把 DataRowVersion 值作为一个参数。例如，下面的代码通过 foreach 语句获得了 DataTable 中每一列的所有值：

```
foreach (DataRow row in ds.Tables["Customers"].Rows )
{
  foreach ( DataColumn dc in ds.Tables["Customers"].Columns )
  {
     Console.WriteLine ("{0} Current  = {1}",dc.ColumnName ,
                                         row[dc,DataRowVersion.Current]);
     Console.WriteLine ("   Default  = {0}",row[dc,DataRowVersion.Defa
ult]);
    Console.WriteLine ("   Original = {0}",row[dc,DataRowVersion.Origi
nal]);
  }
}
```

整个数据行的状态标志 RowState 可以确定在返回数据库时需要对该行进行什么操作。RowState 标志跟踪对 DataTable 所做的所有改变，例如添加行、删除行和改变表中的列。当数据与数据库保持一致时，行的状态标志用于确定应执行什么 SQL 操作。这些标志由 DataRowState 枚举定义，各 DataRowState 枚举值的具体说明如表 17-6 所示。

表 17-6　DataRowState 枚举值

DataRowState 值	说　明
Added	把新数据行添加到 DataTable 的 Rows 集合中。在客户机中创建的所有行都设置为这个值，最终在与数据库保持一致时，会使用 SQL INSERT 语句
Deleted	通过 DataRow.Delete()方法把 DataTable 中的数据行标记为删除。该行仍保存在 DataTable 中，但在屏幕上看不到它（除非显式设置 DataView）。在 DataTable 中标记为已删除的 Rows 将在与数据库保持一致时从数据库中删除
Detached	数据行在创建后立即显示为这个状态，调用 DataRow.Remove()也可以返回这个状态。分立的行不是任何 DataTable 的一部分，因此处于这种状态的行不能使用任何 SQL 语句
Modified	如果列中的值发生了改变，就会修改一行数据
Unchanged	自从最后一次调用 AcceptChanges 以来，数据行都没有发生改变

在 C#中，可以使用索引器来修改 DataRow 中的数据。但如果对数据进行了许多修改，就需要考虑使用 BeginEdit()方法和 EndEdit()方法。

17.7.4　DataSet 数据更新

DataSet 的数据更新包括数据修改、数据添加和数据删除三种方式，上述方式的实现模式是相同的，具体实现过程如下。

1）使用被操作数据填充 DataSet。

2）对 DataSet 的数据进行更新处理。

3）把更新后的 DataSet 数据返回到数据库中。

1．数据修改

使用 DataSet 修改数据的过程比较简单，例如通过如下代码可以修改 Customers 表内的

指定数据：

```
using System.Data.SqlClient;
using System.Data;
using System.Text;
namespace DataRowchuli
{
    class Program
    {
        static void Main(string[] args)
        {
            string source = "server=xxxx;" +
             "uid=sa;pwd=xxxxx;" +
             "database=northwind";
            string select = "SELECT ContactName,CompanyName FROM Customers";
            SqlConnection conn = new SqlConnection(source);
            SqlDataAdapter mm = new SqlDataAdapter(select, conn);
            SqlCommandBuilder cc = new SqlCommandBuilder(mm);
            DataSet nn = new DataSet();
            mm.Fill(nn, "Customers");
            Console.WriteLine("没有修改前是: {0}",
                nn.Tables["Customers"].Rows[12]["CompanyName"]);
            nn.Tables["Customers"].Rows[12]["CompanyName"] = "121212.";
            mm.Update(nn, "Customers");
            Console.WriteLine("修改后是 : {0}",
                nn.Tables["Customers"].Rows[12]["CompanyName"]);
            conn.Close();
            Console.Write("处理完成");
            Console.ReadLine();
        }

    }
}
```

上述代码的实现流程如下。

1）定义连接参数 source，实现和指定数据库 northwind 的连接。

2）定义查询语句参数 select 查询库内 Customer 表的信息。

3）定义 SqlDataAdapter 类对象 mm，用于选择 DataSet 中的数据。

4）定义 DataSet 对象 nn，使用 Fill()方法对表的数据进行处理。

5）通过 Tables 和 Rows 获取原来的数据值。

6）使用 Update()修改为指定的值。

7）通过 WriteLine()输出修改后的值。

将上述代码保存为“光盘:\daima\17\gengxin”，执行后将依次输出显示修改前的值和修改后的值，如图 17-16 所示。

```
没有修改前是: Cactus Comidas para llevar
修改后是 : 121212.
```

图 17-16　输出修改前的值和修改后的值

2．添加数据

使用 DataSet 添加数据行的流程如下。

1）创建新的 DataRow 对象，并在其中添加数据。

2）将上面对象添加到 DataSet 的 Rows 集合内。

3）调用 DataAdapter 的 Update()方法，把添加后的更新分配到数据库中。

3．删除数据

使用 DataSet 内的 Delete()方法可以删除库内指定的数据行。下面通过一个具体的实例来说明使用 DataSet 删除库内指定行的实现方法。

实例 83：使用 DataSet 删除库内指定行

本实例代码保存在“光盘:\daima\17”文件夹内，项目名为 shanchu。实例的功能是输出显示对 Customers 表操作的数据行信息。实例文件 shanchu.cs 的主要代码如下：

```
namespace DeletingData
{
    class Program
    {
        static void Main(string[] args)
        {
            SqlConnection aa = new SqlConnection(
                 @"Server=AAA\BBB;Integrated Security=True;" +
                 "Database=northwind");
            SqlDataAdapter bb = new SqlDataAdapter(
                 "SELECT CustomerID, CompanyName FROM Customers", aa);
            SqlCommandBuilder thisBuilder = new SqlCommandBuilder(bb);
            DataSet cc = new DataSet();
            bb.Fill(cc, "Customers");
            Console.WriteLine("删除前的行数: {0}",
                 cc.Tables["Customers"].Rows.Count);
            DataColumn[] keys = new DataColumn[1];
            keys[0] = cc.Tables["Customers"].Columns["CustomerID"];
            cc.Tables["Customers"].PrimaryKey = keys;
            DataRow dd = cc.Tables["Customers"].Rows.Find("mmm");
            if (dd != null)
            {
                Console.WriteLine("存在删除的数据");
                Console.WriteLine("删除处理 . . .");
                dd.Delete();
                bb.Update(cc, "Customers");
            }
            Console.WriteLine("删除后的行数: {0}",
                 cc.Tables["Customers"].Rows.Count);
```

```
                aa.Close();
                Console.Write("处理完毕！");
                Console.ReadLine();
            }
        }
    }
```

上述实例文件的实现流程如下。

1）定义 SqlConnection 对象 aa，实现和指定数据库 northwin 的连接。
2）定义 SqlDataAdapter 对象 bb，设置查询库内 Customer 表的信息。
3）定义 SqlCommandBuilder 对象 thisBuilder，用于执行 bb 处理语句。
4）使用 Count 获取 Customer 表内原来数据的行数。
5）定义 DataSet 对象 cc，使用 Fill()方法对表的数据进行处理。
6）定义 DataRow 对象 dd，设置删除 Customer 表内 CustomerID 为“mmm”的值。
7）使用 if 语句查询表内是否存在 CustomerID 为“mmm”的值。
8）如果存在 CustomerID 为“mmm”的值，则使用 Delete()方法删除这个值的行。
9）通过 WriteLine()输出删除处理后 Customer 表的数据行数。

执行后将依次输出显示删除前的行数和删除后的行数，如图 17-17 所示。

```
删除前的行数: 94
存在删除的数据
删除处理 . . .
删除后的行数: 93
```

图 17-17　输出删除前的行数和删除后的行数



如果要删除的值在此数据库中不存在，上述实例代码将会引发异常。

17.8　ADO.NET 和 XML

ADO.NET 和 XML 之间存在着十分重要的关系，在 ADO.NET 对象模型内实现了对 XML 的支持。在本节的内容中，将对 ADO.NET 内处理 XML 文件的主要方法的基本知识进行简要介绍。

17.8.1　WriteXml()方法

ADO.NET 通常使用 DataSet 对象来处理 XML 文件，在 DataSet 对象中有七个专用处理方法。其中最为重要的是 WriteXml()，它能够将 DataSet 数据输出显示为 XML 格式。

实例 84：使用 WriteXml()方法实现数据转换

下面将通过一个简单实例的实现过程，向读者介绍使用 WriteXml()方法实现数据转换的过程。本实例保存在“17”文件夹内，项目名为 xmlchuli。实例的功能是使用 WriteXml()方法将 DataSet 对象内的数据转换为 XML 格式文件。实例文件 xmlchuli.cs 的主要代码如下：

```
namespace xmlchuli
{
    class Program
    {
        static void Main(string[] args)
        {

            SqlConnection conn = new SqlConnection(
                @"Server=AAA\BBB;Integrated Security=True;" +
                "Database=northwind");
            SqlDataAdapter aa = new SqlDataAdapter(
                "SELECT CustomerID, CompanyName FROM Customers", conn);
            SqlCommandBuilder bb = new SqlCommandBuilder(aa);
            DataSet cc = new DataSet();
            SqlDataAdapter dd = new SqlDataAdapter(
                "SELECT * FROM Customers", conn);
            SqlDataAdapter ee = new SqlDataAdapter(
                "SELECT * FROM Orders", conn);
            dd.Fill(cc, "Customers");
            ee.Fill(cc, "Orders");
            DataRelation ff = cc.Relations.Add("CustOrders",
                cc.Tables["Customers"].Columns["CustomerID"],
                cc.Tables["Orders"].Columns["CustomerID"]);
            cc.WriteXml(@"e:\Customers.xml");
            Console.WriteLine(@"成功创建了 XML 文件 e:\Customers.xml");
            conn.Close();
            Console.Write("处理完毕！");
            Console.ReadLine();
        }
    }
}
```

上述实例文件的实现流程如下。

1）定义 SqlConnection 对象 conn，实现和指定数据库 northwind 的连接。

2）定义 SqlDataAdapter 对象 aa，设置查询库内 Customer 表的信息。

3）定义 SqlCommandBuilder 对象 bb，用于执行 aa 处理语句。

4）定义 DataSet 对象 cc。

5）定义 SqlDataAdapter 对象 dd，设置查询 Customer 表的信息。

6）定义 SqlDataAdapter 对象 ee，设置查询 Orders 表的信息。

7）使用 Fill()方法对上述表的数据进行处理。

8）定义 DataRelation 对象 ff，使用方法 Add()设置表的输出数据。

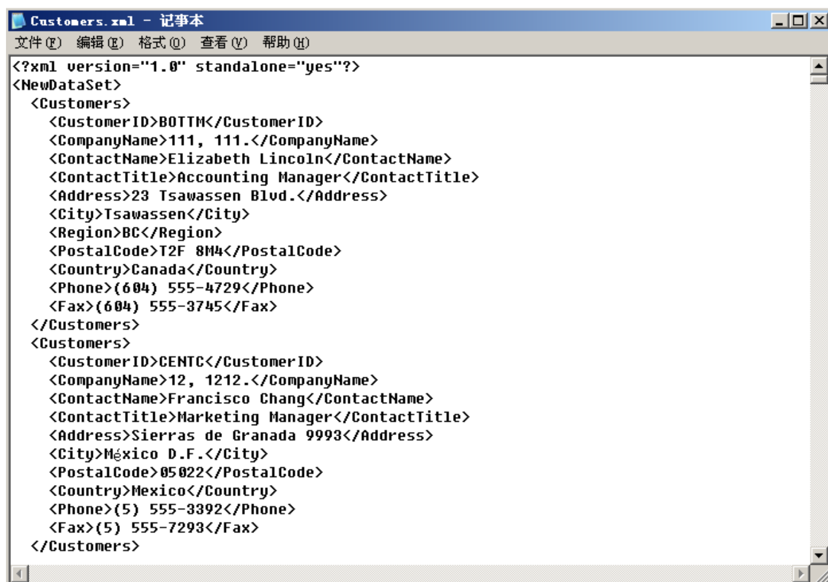
9）通过 WriteXml()方法将表数据转换为 XML 格式文件。

执行后将输出转换成功提示，如图 17-18 所示。

成功创建了XML文件 e:\Customers.xml

图 17-18　输出转换成功提示

上述程序处理完毕后将在指定路径“e:\”生成 XML 格式文件“Customers.xml”，如图 17-19 所示。


Customers.xml - 记事本
文件(F)　编辑(E)　格式(O)　查看(V)　帮助(H)

```
<?xml version="1.0" standalone="yes"?>
<NewDataSet>
  <Customers>
    <CustomerID>BOTTM</CustomerID>
    <CompanyName>111, 111.</CompanyName>
    <ContactName>Elizabeth Lincoln</ContactName>
    <ContactTitle>Accounting Manager</ContactTitle>
    <Address>23 Tsawassen Blvd.</Address>
    <City>Tsawassen</City>
    <Region>BC</Region>
    <PostalCode>T2F 8M4</PostalCode>
    <Country>Canada</Country>
    <Phone>(604) 555-4729</Phone>
    <Fax>(604) 555-3745</Fax>
  </Customers>
  <Customers>
    <CustomerID>CENTC</CustomerID>
    <CompanyName>12, 1212.</CompanyName>
    <ContactName>Francisco Chang</ContactName>
    <ContactTitle>Marketing Manager</ContactTitle>
    <Address>Sierras de Granada 9993</Address>
    <City>México D.F.</City>
    <PostalCode>05022</PostalCode>
    <Country>Mexico</Country>
    <Phone>(5) 555-3392</Phone>
    <Fax>(5) 555-7293</Fax>
  </Customers>
```

图 17-19　生成的 Customers.xml


多学一招

在 XML 文件中的数据量不大，或者数据架构不复杂的情况下，使用这种方法读取的速度很快，但是对于大文件的读取速度就会明显地降下来，将会影响程序的运行。

使用 DataSet 的 WriteXml()方法，可以将 DataSet 中的数据和架构写入到 XML 文件中保存。通常在 WriteXml()方法中使用的参数是 String 类型的，但是 WriteXml()方法还有另外的一种 Stream 参数类型。两种 WriteXml()方法的完整写法如下：

```
DataSet.WriteXml(String filename)
DataSet.WriteXml(Stream filename)
```

17.8.2　ReadXml()方法

除了 17.8.1 中的 WriteXml()方法外，DataSet 可以使用 ReadXml()创建一个 DataSet 对象，并且这个对象内的数据是用指定 XML 文件数据填充的。ReadXml()能够在 DataSet 内创建一个 DataTable 对象，并且这个 DataTable 是以 XML 文件的根元素命名的。

实例 85：使用 ReadXml()方法读取 XML 文件中的数据

下面将通过一个简单实例的实现过程，向读者介绍使用方法 ReadXml()实现读取 XML 文件中的数据的过程。本实例保存在“光盘:\daima\17”文件夹内，项目名为 ReadXml chuli。实例的功能是使用 WriteXml()读取 XML 文件中的内容。实例文件 ReadXmlchuli.cs 的主要代码如下：

```
namespace ReadXmlchuli {
```

```
        class Program
        {
            static void Main(string[] args)
            {
                DataSet aa = new DataSet();
                aa.ReadXml(@"e:\Customers.xml");
                foreach (DataRow custRow in aa.Tables["Customers"].Rows)
                {
                  Console.WriteLine("Customer ID: " + custRow["CustomerID"] +
                                   " Name: " + custRow["CompanyName"]);
                }
                Console.WriteLine("以XML文件数据创建表{0}完毕",
                                aa.Tables[0].TableName);
                Console.Write("处理完毕！");
                Console.ReadLine();
            }
        }
    }
```

上述实例文件的实现流程如下

1）定义 DataSet 对象 aa，然后使用 ReadXml()读取指定的 XML 文件“e:\Customers.xml”的数据。

2）使用 foreach 语句将 XML 元素转换为 DataSet 的对象内容。

3）通过 WriteLine()方法输出 DataSet 的对象内的数据。

执行后将输出创建 DataSet 对象的数据，如图 17-20 所示。

```
Customer ID: BOLID Name: 888, 888.
Customer ID: aaa   Name: aaa.
Customer ID: ALFKI Name: Alfreds Futterkiste
Customer ID: ANATR Name: Ana Trujillo Emparedados y helados
Customer ID: ANTON Name: Antonio Moreno Taquería
Customer ID: AROUT Name: Around the Horn
Customer ID: BERGS Name: Berglunds snabbk?p
Customer ID: BSBEV Name: B's Beverages
Customer ID: CHOPS Name: Chop-suey Chinese
Customer ID: COMMI Name: Comércio Mineiro
Customer ID: CONSH Name: Consolidated Holdings
Customer ID: WANDK Name: Die Wandernde Kuh
Customer ID: DRACD Name: Drachenblut Delikatessen
```

图 17-20　输出创建 DataSet 对象的数据



多学一招

单纯用 DataSet 的 ReadXml()方法读取 XML 对于小数据量来说效率很高，但是对大数据量的 XML 来说就有些力不从心了。ReadXML()默认使用 XmlReadMode.Auto 方式读取，这就给大数据量留下了低效率的伏笔。因为如果 XML 没有 Scheme，DataSet 就会自己推算 XML 的结构，然后再加载，这就是低效的主要原因。让 DataSet 推算不如帮它算，因为 DataSet 已经提供了一个 ReadXmlSchema()方法。

17.9 ADO.NET 和 SQL

SQL 是通用的数据库处理语言，可以实现对指定数据库数据的查询、修改、添加和删除等操作。在本节的内容中，将对 ADO.NET 内使用 SQL 语言的基本知识进行简要介绍。

17.9.1 更新操作命令

在 SQL 中有四种更新操作命令，分别是 SELECT（查询）、UPDATE（修改）、INSERT（添加）和 DELETE（删除）。在 SqlCommandBuilder 对象中使用上述操作命令，可以实现对数据库数据的更新处理。SqlCommandBuilder 对象中的主要处理方法如下。

- GetUpdateCommand()方法：用于数据修改。
- GetInsertCommand()方法：用于添加某数据。
- GetDeleteCommand()方法：用于删除某数据。

实例 86：使用 SqlCommandBuilder 对象方法实现数据处理

下面将通过一个简单实例的实现过程，向读者介绍使用 SqlCommandBuilder 对象方法实现数据处理过程。本实例代码保存在“光盘:\daima\17”文件夹内，项目名为 sqlchuli。实例的功能是将对应处理方法转换为等效的 SQL 处理语句。实例文件 sqlchuli.cs 的主要代码如下：

```
namespace sqlchuli
{
   class Program
   {
      static void Main(string[] args)
      {
         SqlConnection mm = new SqlConnection(
             @"Server=AAA\BBB;Integrated Security=True;" +
            "Database=northwind");
         mm.Open();
         SqlDataAdapter aa = new
           SqlDataAdapter("SELECT CompanyName from Customers", mm);
         SqlCommandBuilder bb = new SqlCommandBuilder(aa);
         Console.WriteLine("查询语句是:\n{0}\n",
                        aa.SelectCommand.CommandText);
         SqlCommand updateCommand = bb.GetUpdateCommand();
         Console.WriteLine("修改语句是:\n{0}\n",
                        updateCommand.CommandText);
         SqlCommand insertCommand = bb.GetInsertCommand();
         Console.WriteLine("添加语句是:\n{0}\n",
                        insertCommand.CommandText);
         SqlCommand deleteCommand = bb.GetDeleteCommand();
         Console.WriteLine("删除语句是:\n{0}",
                        deleteCommand.CommandText);
         mm.Close();
```

```
                Console.Write("处理完毕!! ");
                Console.ReadLine();
            }
        }
    }
```

上述实例文件的实现流程如下。

1）定义 SqlConnection 对象 mm，实现和指定数据库 northwind 的连接。

2）定义 SqlDataAdapter 对象 aa，设置查询库内 Customer 表的信息。

3）定义 SqlCommandBuilder 对象 bb，用于执行 aa 处理语句。

4）通过 WriteLine 输出和上述处理等效的对应 SQL 语句。

5）调用 GetUpdateCommand()方法，通过 WriteLine()输出和 GetUpdateCommand 处()理等效的对应 SQL 语句。

6）调用 GetInsertCommand()方法，通过 WriteLine()输出和 GetInsertCommand()处理等效的对应 SQL 语句。

7）调用 GetDeleteCommand()方法，通过 WriteLine()输出和 GetDeleteCommand()处理等效的对应 SQL 语句。

执行后输出显示和 SqlCommandBuilder 对象方法等效的 SQL 语句，如图 17-21 所示。

```
查询语句是:
SELECT CompanyName from Customers

修改语句是:
UPDATE [Customers] SET [CompanyName] = @p1 WHERE (([CompanyName] = @p2))

添加语句是:
INSERT INTO [Customers] ([CompanyName]) VALUES (@p1)

删除语句是:
DELETE FROM [Customers] WHERE (([CompanyName] = @p1))
```

图 17-21　输出等效 SQL 语句


多学一招

通过 SqlCommandBuilder.DeriveParameters()方法从数据库里获取 StoredProcedure 的参数，这样能降低附加存储过程参数的代码量，提高开发效率，但是有可能会对数据库性能造成影响。但是只要不是频繁调用，DeriveParameters()不会影响性能。另外如果需要经常调用的是存储过程，就可以调用 DeriveParameters 一次，然后就缓存用到的参数，这样会将影响降到最低。

17.9.2　直接执行 SQL 命令

ADO.NET 可以使用 SqlCommand 对象和 OleDbCommand 对象直接执行 SQL 处理命令，这样就不必编写 C#代码即可实现对数据库的处理。在下面的内容中，将简要介绍直接使用 SQL 命令的方法。

1. 数据查询

使用 SqlCommand 的 ExecuteScalar()方法可以查询数据库内的某条信息。例如在下面的代码中，直接使用 SQL 语句查询显示了表 Customers 中数据的行数。

```
namespace ExecuteScalarExample
{
    class Program
    {
        static void Main(string[] args)
        {
            SqlConnection mm= new
            SqlConnection(@"Server=AAA\BBB;Integrated Security=True;" +
                    "Database=northwind");
            mm.Open();
            SqlCommand aa = mm.CreateCommand();
            aa.CommandText = "SELECT COUNT(*) FROM Customers";
            Object bb = aa.ExecuteScalar();
            Console.WriteLine("行数 = {0}\n", bb);
            mm.Close();
            Console.Write("处理完毕!! ");
            Console.ReadLine();
        }
    }
}
```

上述代码的实现流程如下。

1）定义 SqlConnection 对象 mm，实现和指定数据库 northwind 的连接。

2）定义 SqlCommand 对象 aa，然后使用 CommandText 设置执行的 SQL 查询语句。

3）通过 ExecuteScalar()方法执行查询语句。

4）通过 WriteLine 输出表 Customers 内数据的行数。

上述代码保存在“光盘:\daima\17\zhijie”，执行将输出显示表 Customers 内数据的行数，如图 17-22 所示。

行数 = 93

图 17-22　输出显示行数

2. 数据更新

使用 SqlCommand 的 ExecuteNonQuery()方法，可以更新库内指定行的信息。

实例 87：使用 ExecuteNonQuery()方法实现 SQL 处理

下面将通过一个简单实例的实现过程，向读者介绍使用 ExecuteNonQuery()方法实现直接 SQL 处理的过程。本实例保存在“17”文件夹内，项目名为 zhijiegengxin。本实例文件的功能是将表 Products 内 SupplierID 值为“光盘:\daima\3”的行的 UnitPrice 值增加一倍，然后输出被更新处理后的行数。

实例文件 zhijiegengxin.cs 的主要代码如下：

```
namespace zhijiegengxin
{
    class Program
```

```
        {
            static void Main(string[] args)
            {
                SqlConnection mm = new SqlConnection(
                    @"Server=AAA\BBB;Integrated Security=True;" +
                    "Database=northwind");
                mm.Open();
                SqlCommand aa = mm.CreateCommand();
                aa.CommandText = "UPDATE Products SET " +
                    "UnitPrice=UnitPrice*2 WHERE SupplierId=3";
                int cc = aa.ExecuteNonQuery();
                Console.WriteLine("Rows Updated = {0}\n", cc);
                mm.Close();
                Console.Write("处理完毕!!! ");
                Console.ReadLine();
            }
        }
    }
```

上述实例文件的实现流程如下。

1）定义 SqlConnection 对象 mm，实现和指定数据库 northwind 的连接。

2）定义 SqlCommand 对象 aa，然后使用 CommandText 设置执行的 SQL 更新语句。

3）通过 ExecuteNonQuery()方法执行更新语句。

4）通过 WriteLine 输出表 Products 内数据被更新的行数。

执行后将输出显示表 Products 内被更新处理的行数，如图 17−23 所示。

```
Rows Updated = 3
```

图 17−23　输出被更新的行数


多学一招

在 ADO.NET 中经常需要跟各种数据库打交道，在不使用存储过程的情况下，使用参数化 SQL 语句一定程度上可以防止 SQL 注入，同时对一些较难赋值的字段（如在 SQL Server 中赋值 Image 字段，在 Oracle 中赋值 Clob 字段等）使用参数化 SQL 语句很容易就能赋值，所以笔者经常在 ADO.NET 中使用参数化 SQL 语句，在此也建议读者尽量使用参数化 SQL 语句。

在参数化 SQL 中参数名的格式与其在存储过程中生成存储过程参数一致，例如在 Oracle 中存储过程参数一律以“:”开头，在 MS SQL Server 中存储过程参数一律以“@”开头，而在 MySQL 中存储过程（MySQL 从 5.0 以后版本支持存储过程）参数一律以“？”开头，所以在参数化 SQL 语句中参数名有些不一样。

17.10　疑难问题解析

本章的内容详细介绍了 C#在数据库和 ADO.NET 操作方面的基本知识。本节将对本章中比较难以理解的问题进行讲解。

读者疑问：请详细介绍概念模型的基本知识。

解答：概念模型是对信息世界的建模，它可以用 E-R 图（实体–联系图）来描述世界

的概念模型。E-R 图提供了表示实体型、属性和联系的方法。

- 实体型：用矩形表示，框内写实体名称。
- 属性：用椭圆表示，框内写属性名称。
- 联系：用菱形表示，框内写联系名称。

例如，图 17–24 为实体–属性图。

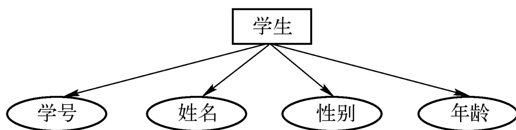


图 17–24　实体–属性图

图 17–25 描述了实体–联系图。

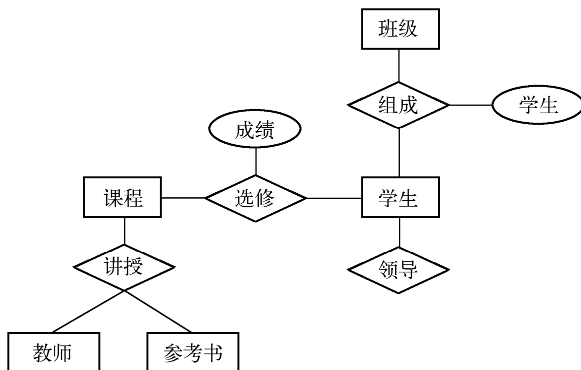


图 17–25　实体–联系图

读者疑问：DataSet 和 DataReader 功能类似，请问在具体使用时应该怎样选择？

解答：当设计应用程序时，要考虑应用程序所需功能的等级，以确定使用 DataSet 或者是 DataReader。一般来说，具体规则如下。

1）要通过应用程序执行以下操作，就要使用 DataSet。

- 在结果的多个离散表之间进行导航。
- 操作来自多个数据源（例如，来自多个数据库、一个 XML 文件和一个电子表格的混合数据）的数据。
- 在各层之间交换数据或使用 XML Web 服务。与 DataReader 不同的是，DataSet 能传递给远程客户端。
- 重用同样的记录集合，以便通过缓存获得性能改善（例如排序、搜索或筛选数据）。
- 每条记录都需要执行大量处理。对使用 DataReader 返回的每一行进行扩展处理会延长服务于 DataReader 的连接的必要时间，这影响了性能。
- 使用 XML 操作对数据进行操作，例如可扩展样式表语言转换（XSLT 转换）或 XPath 查询。

2）对于下列情况，要在应用程序中使用 DataReader。

- 不需要缓存数据。
- 要处理的结果集太大，内存中放不下。
- 需要以仅向前、只读方式快速访问数据。

在填充 DataSet 时，DataAdapter 使用 DataReader。因此，使用 DataAdapter 取代 DataSet 提升的性能，会节省 DataSet 占用的内存和填充 DataSet 需要的循环。一般来说，此性能提升只是象征性的，因此，设计决策应以所需功能为基础。



职场点拨——寻找更好的工作

在职场中的 IT 精英们肯定会有寻找更好的工作的想法，但是换工作是既有利的也是有弊的。企业培养一个程序员不容易，辛苦带起来一个人，熟悉了业务，掌握了技术，这时候走人，损失最大的当然是企业，花时间培养人和熟悉业务也是需要成本的。对程序员来说，跳槽几乎是利大于弊，待遇肯定立刻提高，除非有其他想法和目标。

换工作对于技术人员来说并不是坏事，原因有二。

1）一般的 IT 公司都有自己常用的模式，该模式经过一个项目之后，就可以基本掌握，相关覆盖的知识、架构等大概也可以了解，此时可以换个环境寻找更高的发展。

2）跳槽相当于变相的升职，这个可以从你的简历中体现出来。

在换工作时，是否能够获得满意的岗位是需要技巧的。不同的级别会有不同的技巧，接下来将一一讲解。

（1）初级程序员

当做完一个项目时，初级程序员会了解这个项目的整个流程，此时可以在简历中填写中级程序员的申请，把很多中级程序员做的事情写到履历里（前提是要了解这些），跳槽的时候，目标自然就是中级程序员，而招聘公司看到简历时也会觉得合适。

（2）中级程序员

需要在项目中了解高级程序员的工作范围，并不要求全部掌握，但要能表达出来，这个很重要。比如后台的设计模式、软件架构、接口设计等，把这些写到履历中，给自己定位成高级程序员，自然而然，高级程序员的职位会找过来。

（3）高级程序员

所需要了解的不仅仅是程序设计，还包括整个项目的运作和管理流程。包括项目管理、系统架构（软硬件）、系统集成等，整个环节不一定都要会，但需要知道是什么，比如，什么是交换机，什么是硬件负载均衡设备，什么是反向代理，什么是缓存服务器，什么是 Web 服务器，什么是集群、负载均衡、分布式、数据库优化、大数据存储、高并发访问等，都是需要了解的，面试的时候能表达出来，那么就成功了。同样可以把这些写到履历中，给自己定位架构师或项目经理，更新简历后，猎头会找过来。

（4）系统架构师

既然选择了架构师的角色，那么肯定是向技术方向发展了。技术总监、研发总监甚至

CTO 就是目标。技术总监需要负责整个公司的技术部运作，包括对人员的管理、绩效考核、各语言组间的协调、各项目间的协调，各部门间的协调，除此之外，还需要考虑所运营的项目如何发展得更好，网站如何才能更加优化，产品如何能更上一个层次，公司的技术发展如何规划，各种方案如何快速地编写和实施，如何与老板打交道等，都是需要掌握的。

（5）项目经理

项目经理分两种，一种是团队领导者的角色，需要很强的技术；另一种是负责招标、流程控制的偏商务角色，要懂技术。发展到这个层次的，应该不用笔者来告诉他们出路在哪里了，这样的人一般都有自己的规划，但凡事都有例外，如果团队领导者没有规划或发展迷茫的，角色可以重点把项目管理、人力资源、系统架构等环节再强化一下，紧跟当前发展形势学习新知识；偏商务角色的，可以考虑向总经理、CIO、CEO 等方向努力。到这个层次的，需要的不仅仅是知识，更多的是一种理念和个人魅力。

第 18 章 DataGrid 和数据绑定

在计算机项目中，几乎所有的应用项目都是基于数据库的。无论是桌面项目，还是 Web 项目。小到公司的网站商品宣传和工资统计，大到省级甚至国家级的计费系统，几乎都使用了数据库来存储这些海量的信息。在前面的内容中，已经了解了 C#内使用 ADO.NET 连接数据库的方法。其实在实际项目开发过程中，还可以使用 DataGrid 获取数据库的数据，并实现数据的绑定。通过本章能学到如下知识。

- 通过 Visual Studio2010 实现数据库项目。
- DataGrid 控件。
- 数据绑定。
- 职场点拨——程序员创业的致命错误。

2008 年 XX 月 XX 日，小雪

我最近压力依旧特别大，很多地方都需要花钱。今天听大学同学 A 说：同学 B 创业有成，买房、买车，并举办了一场隆重的婚礼。我很羡慕。我也想通过创业来改变自己的命运。



一问一答

小菜：我想个人创业，不知是否可行。

Wisdom：很不错。但是，创业成功会使你得到想要的一些东西，万一失败，你也会失去很多东西，并对你的自信心会带来很大的打击。所以你得三思而后行，建议你参考本章最后的“程序员创业的致命错误”。

小菜：嗯，我会好好考虑的！言归正传，本章的 DataGrid 和数据绑定很重要吗？

Wisdom：前面已经说过，数据库在一定程度上改变了软件产业的命运。但是数据究竟有什么作用呢？其实就是显示数据。C#中通常通过 DataGrid 和数据绑定来实现数据的显示和管理，所以两者的重要性不言而喻。

18.1 通过 Visual Studio 2010 实现数据库项目

通过 Visual Studio 2010 不但可以进行应用程序设计，而且能够和数据库相结合而实现数据库连接，并能对数据库的数据进行相关操作。在本节的内容中，将向读者简要介绍在 Visual Studio 2010 内进行相关数据库操作的基本知识。

18.1.1　建立数据库连接

利用 Visual Studio 2010 可以很容易地实现和数据库的连接，具体实现步骤如下。

1）在 Visual Studio 2010 中新建一个名为“VS”的控制台项目，如图 18-1 所示。

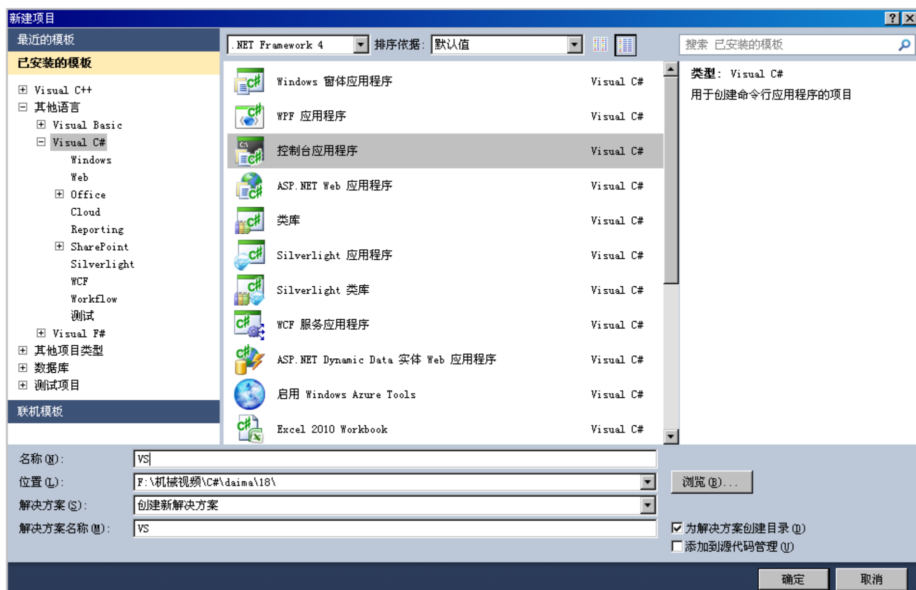


图 18-1　新建一个控制台项目

2）依次单击项目工具栏中的“工具”｜“连接到数据库”命令，弹出“选择数据源”对话框，如图 18-2 所示。

3）在“服务器名”中输入要连接的数据库服务器名称“XXX”，然后分别输入登录的用户名和密码，在“选择或输入一个数据库名”的下拉列表中选择数据库“Northwind”，如图 18-3 所示。

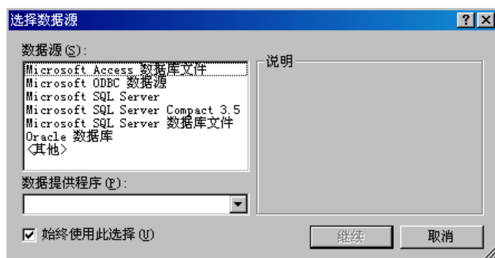


图 18-2　“选择数据源”对话框

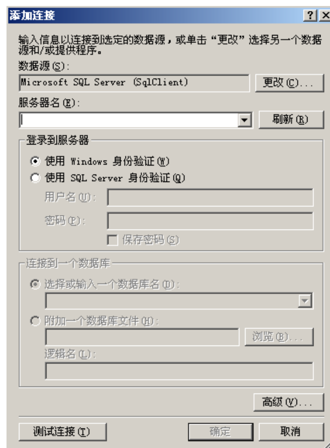


图 18-3　输入连接参数

4）单击“测试连接”按钮后弹出连接成功对话框，如图 18-4 所示。

5）单击“确定”按钮来到 Visual Studio 2010 界面，在“服务器资源管理器”中将出现上面建立的数据库连接，如图 18-5 所示。



图 18-4　连接成功对话框



图 18-5　“服务器资源管理器”中的连接

在 Visual Studio 2010 “服务器资源管理器”中打开连接的数据库，将展开连接数据库的树形结构，如图 18-6 所示。可以看出连接树形结构和 SQL Server 的树形结构完全相同，开发人员可以直接在 Visual Studio 2010 中对连接的数据库进行查看等操作，如图 18-7 所示。

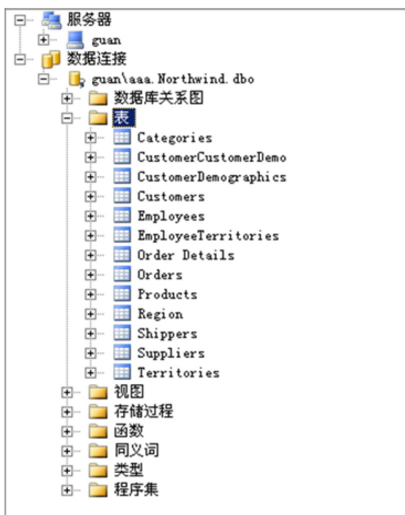


图 18-6　连接数据库的树形结构

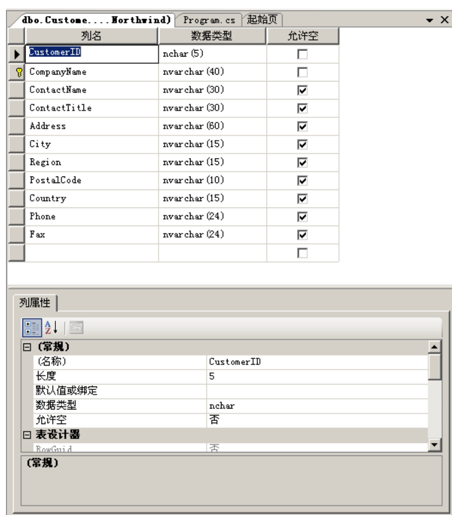


图 18-7　查看数据库的设计结构

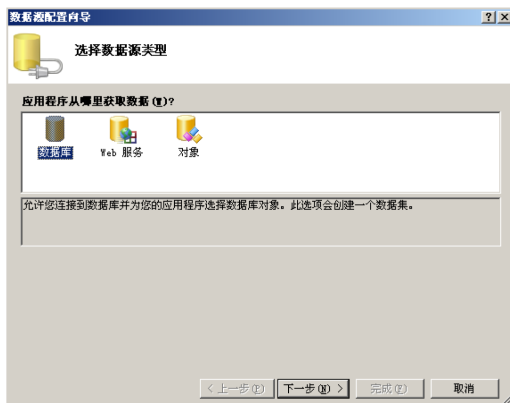
18.1.2　建立数据源

使用 Visual Studio 2010 建立和数据库的连接后，应用程序还是不能实现对数据库的操作。接下来需要为应用程序建立数据源，使程序能够对连接的数据库进行操作。

下面介绍 Visual Studio 2010 为应用程序创建数据源的实现流程，具体操作步骤如下。

1）在 Visual Studio 2010 中依次单击工具栏中的“数据”｜“添加新数据源”命令，弹出“数据源配置向导”对话框，如图 18-8 所示。

2）选择“数据库”选项后单击“下一步”按钮来到“选择连接”界面，选择 18.1.1 节连接建立的数据库，然后在“是否包含敏感数据”选项中选择“是”，如图 18-9 所示。



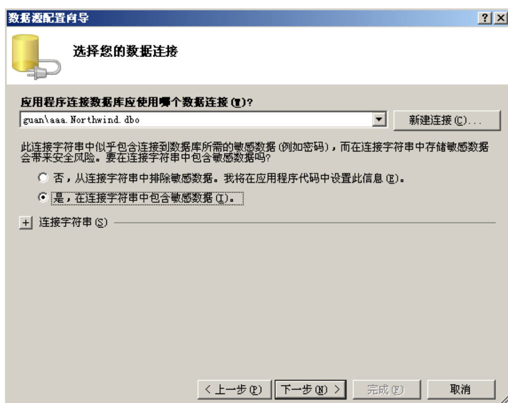


图 18-8　“数据源配置向导”对话框　　　　图 18-9　选择数据源连接

3）单击“下一步”按钮来到“将连接字符串保存到应用程序配置文件中”界面，选择“是，将连接保存为（Y）:”选项，如图 18-10 所示。

4）单击“下一步”按钮来到“选择数据库对象”界面，选择前三个选项，如图 18-11 所示。

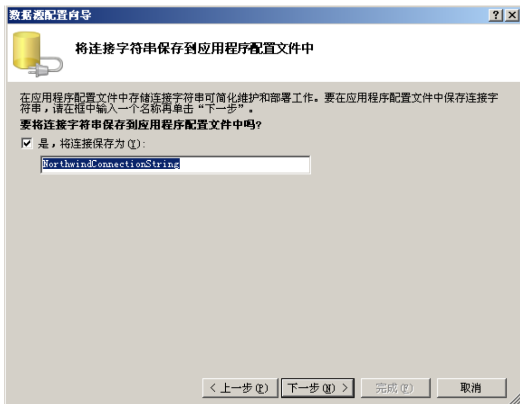
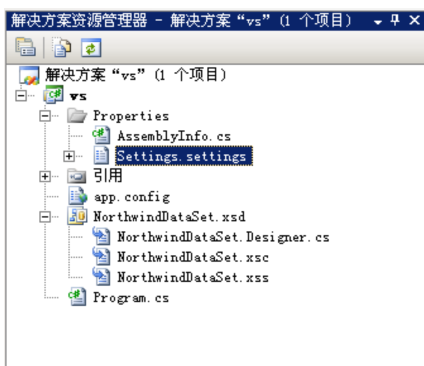




图 18-10　选择保存选项　　　　图 18-11　选择数据库对象

5）单击“完成”按钮完成应用程序数据源的创建工作，在 Visual Studio 2010 的“资源管理器”中将自动创建和数据库相关的文件，如图 18-12 所示。在 Visual Studio 2010 的“数据源”中将显示对应的数据源列表，如图 18-13 所示。



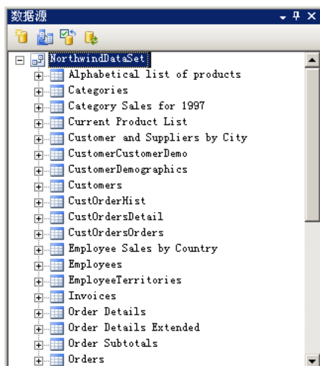


图 18-12　相关的数据库文件　　　　图 18-13　数据源列表

资源管理器中的文件 app.config 设置了和连接数据库的连接参数，具体代码如下：

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <configSections>
    </configSections>
    <connectionStrings>
       <add name="vs.Properties.Settings.NorthwindConnectionString"
          connectionString="Data Source=GUAN\AAA;Initial Catalog=North
wind;User ID=sa;Password=888888"
          providerName="System.Data.SqlClient" />
    </connectionStrings>
</configuration>
```

18.2　DataGrid 控件

DataGrid 控件具有很好的灵活性，它的数据源可以是 Array、DataTable、DataView 和 DataSet 类，或者是实现 IListSource 或 IList 接口的组件。DataGrid 控件可以显示数据的各种视图，可以通过调用方法 SetDataBinding 来显示 DataSet 类中的数据。

18.2.1　DataGrid 基础

DataGrid 能够非常灵活地显示数据源中的数据。除了调用方法 SetDataBinding()时参数可以是 DataSet 和要显示的表名外，还可以将下述任何一个数据源作为参数调用该方法。

- 数组。
- DataTable。
- DataView。
- DataSet 或 DataViewManager。
- 实现 IListSource 接口的组件。
- 实现 IList 接口的组件。

1．显示数组数据

实现显示数组数据的功能比较简单，只需将创建的数组填充一些数据，然后在 DataGrid 控件上调用 SetDataSource（array，null）即可。看下面的代码：

```
string[] stuff = new string[] {"One", "Two", "Three"};
dataGrid.SetDataBinding(stuff, null);
```

SetDataBinding 带有两个参数，第一个参数是数据源，在本例中就是数组，第二个参数是 null，除非数据源是一个 DataSet 或 DataViewManager 类，此时这个参数应是要显示的表名。

2．DataTable

可以通过如下两种方式在 DataGrid 控件中显示 DataTable。

- 如果 DataTable 是独立的，则调用 SetDataBinding（DataTable，null）实现。

- 如果在 DataSet 中包含 DataTable，则调用 SetDataBinding（DataSet, "<Table Name >"）实现。

3．显示 DataView 数据

在 DataView 中，实现了对 DataTable 中数据的过滤和排序操作。当从数据库中选择数据时，用户可以单击列标题，并对数据进行排序。并且可以只过滤要显示在某些行中的数据，例如用户修改过的所有数据。DataView 允许限制要显示给用户的数据行，但不允许限制 DataTable 中的数据列。

例如，根据现有的 DataTable 创建 DataView 的代码如下：

```
DataView dv = new DataView(dataTable);
```

创建好后，就可以改变 DataView 上的设置，当该视图显示在 DataGrid 中时，这些设置会影响要显示的数据，以及允许对这些数据进行的操作，具体说明如下。

- 设置 AllowEdit = false 表示在数据行上禁用所有列的编辑功能。
- 设置 AllowNew = false 表示禁用新行功能。
- 设置 AllowDelete = false 表示禁用删除行的功能。
- 设置 RowStateFilter 只显示指定状态的行。
- 设置 RowFilter 可过滤数据行。



DataView 不允许修改要显示的数据列，只允许修改要显示的数据行。

（1）通过数据过滤数据行

创建好 DataView 后，可以通过设置 RowFilter 属性来改变视图中的数据。RowFilter 属性是一个字符串，可用做按照给定条件过滤数据的一种方式，该字符串的值就是过滤条件。其语法类似于一般 SQL 中的 WHERE 子句，但主要是对已经从数据库中选择出来的数据进行操作。

过滤子句的常用示例如表 18-1 所示。

表 18-1　过滤子句的常用示例

子　句	说　明
UnitsInStock > 50	只显示 UnitsInStock 列大于 50 的行
Client = 'Smith'	只返回给定客户的记录
County LIKE 'C*'	返回 County 字段以 C 开头的所有记录，例如返回 Cornwall、Cumbria、Cheshire 和 Cambridgeshire 所在的行，可以使用%表示匹配一个字符的通配符，而*表示匹配 0 个或多个字符的通配符

在运行环境中，应尽可能在过滤表达式中使用与源列相匹配的数据类型。但如果提供了一个无效的过滤字符串，就会产生 EvaluateException。

（2）根据状态过滤数据行

DataView 中的每一行都有一个定义好的行状态，各状态值的具体说明如表 18-2 所示。

表 18-2 DataView 行状态值

DataViewRowState	说 明
Added	新创建的所有行
CurrentRows	除了被删除的行以外的其他行
Deleted	最初被选中，且已经删除的行——不显示已经删除的新建行
ModifiedCurrent	列出所有已被修改的行，并显示这些行的当前值
ModifiedOriginal	列出所有已被修改的行，但显示这些行的初值，而不是当前值
OriginalRows	最初从数据源中选中的所有行（不包括新行）显示列的初值，即使进行了修改，不显示当前值
Unchanged	没有修改的行

过滤器不仅可以用于可视化的行，还可以用于这些行中列的状态。假如用户更新了数据行中的一列，该行就会在选择 ModifiedOriginal 或 ModifiedCurrent 时显示出来，但是其实际值可以是从数据库中选择出来的初值（如果选择了 ModifiedOriginal），或者 DataColumn 中的当前值（如果选择了 ModifiedCurrent）。

（3）对数据行进行排序

除了过滤数据外，有时还需要对 DataView 中的数据进行排序。可以在 DataGrid 控件中单击列标题，这会按照升序或降序的顺序对该列进行排序。但是控件只能对一列进行排序，而底层的 DataView 可以对多个列进行排序。

在对数据列进行排序时，既可以单击列的标题，也可以通过代码排序，DataGrid 会显示一个箭头位图，表示对哪一列进行排序。DataView 支持对列进行升序或降序排序，默认为升序。如果选择对 DataView 中的多个列进行排序，DataGrid 就不会显示任何排序箭头。

网格中的每一列都是类型化的，其排序顺序不是基于列的字符串表示，而是基于实际的数据。如果 DataGrid 有一个日期列，要对它进行排序，网格就会按日期来进行排序，而不是按日期字符串来进行排序。

4．显示 DataSet 类数据

DataGrid 主要用于显示 DataSet 中的数据，虽然它一次只能显示一个 DataTable，但是可以浏览 DataSet 中的具体关系。在 DataGrid 控件的右上角有两个新图标，使用箭头可以允许用户导航回到父行，来显示上一页的内容。标题行显示父记录的细节，单击另一个按钮会隐藏或显示该箭头。

5．DataViewManager 中显示数据

在 DataViewManager 中显示的数据与在 DataSet 中显示的相同。当为 DataSet 创建 DataViewManager 时，会为每个 DataTable 创建一个单独的 DataView，根据过滤条件或者行的状态改变显示出来的行。即使不希望过滤数据，也可以把 DataSet 包装到 DataViewManager 中进行显示，只有这样才能在修改源代码时使用更多的选项。

18.2.2 使用 DataGridView

DataGridView 在 Visual Studio 的工具箱中，通过 DataGridView 可以在窗体中显示连接数据源的详细数据。在本节的内容中，将通过一个简单实例来说明 DataGridView 的具体使用流程。

实例 88：使用 DataGridView

本实例代码保存在“光盘:\daima\18”文件夹内，项目名为 chuli。项目实例的功能是创建一个窗体项目，然后通过 DataGridView 控件显示数据库 Northwind 中表 Customers 中四个列的数据。本实例的实现步骤如下所示。

1）使用 Visual Studio 2010 创建一个 Windows 应用程序，如图 18-14 所示。

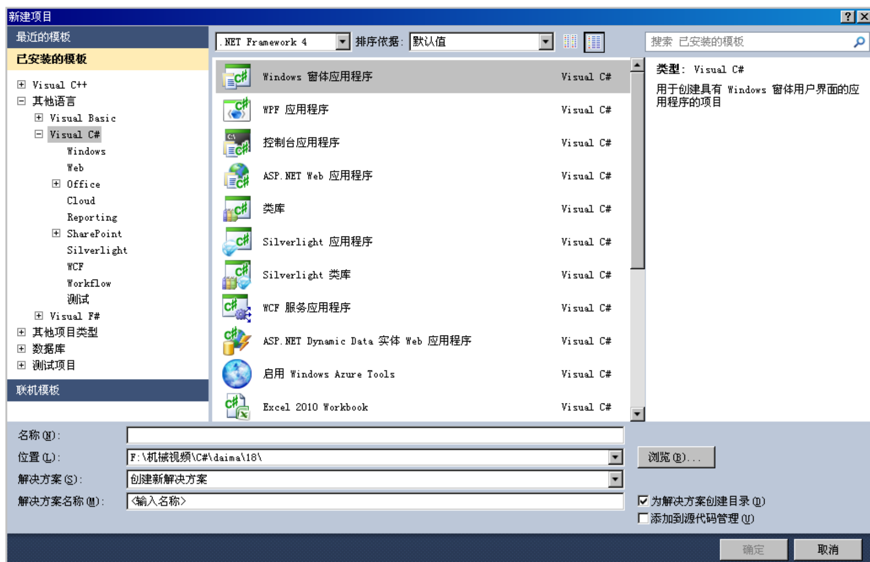


图 18-14　新建 Windows 应用程序

2）建立和数据库 Northwind 的连接，如图 18-15 所示。

3）添加数据库 Northwind 的应用程序的数据源，如图 18-16 所示。

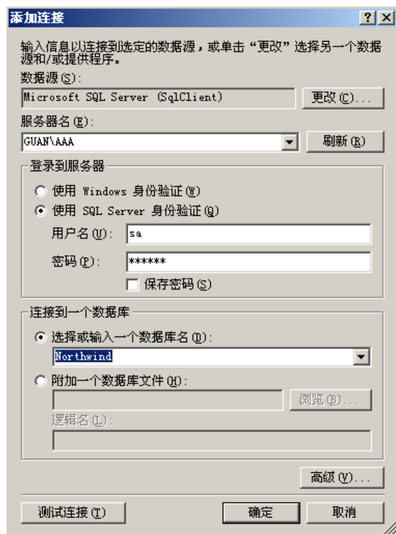


图 18-15　建立数据库连接

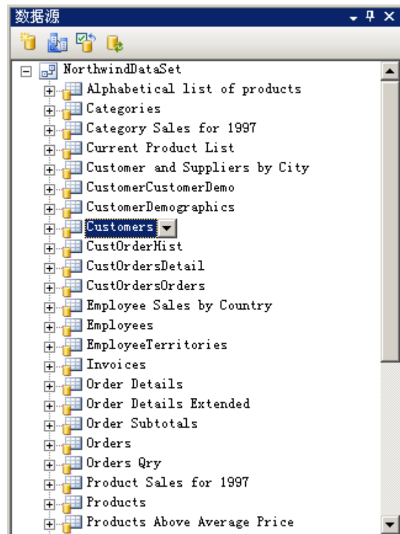


图 18-16　建立应用程序数据源

4）调整 Form1 窗体的大小，使之能够放开 DataGridView 控件，如图 18-17 所示。

5）从 Visual Studio 2010 “工具箱”中拖入到 Form1 中一个 DataGridView 控件，并调整控件的大小，如图 18-18 所示。

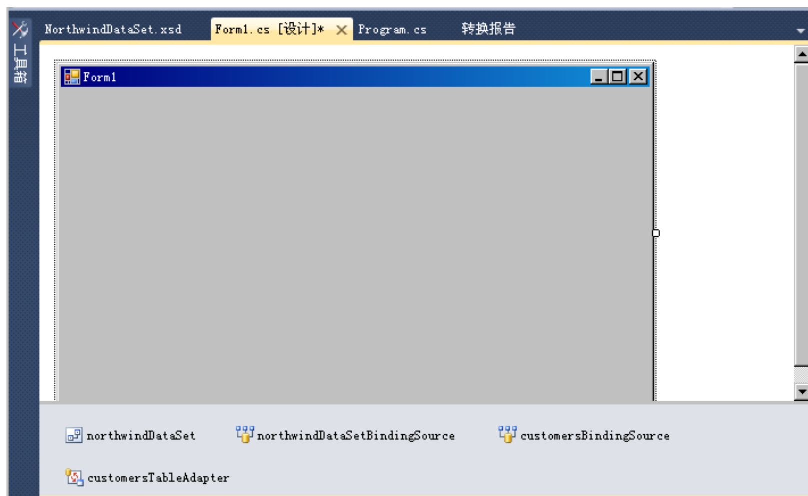


图 18-17　调整 Form 窗体的大小

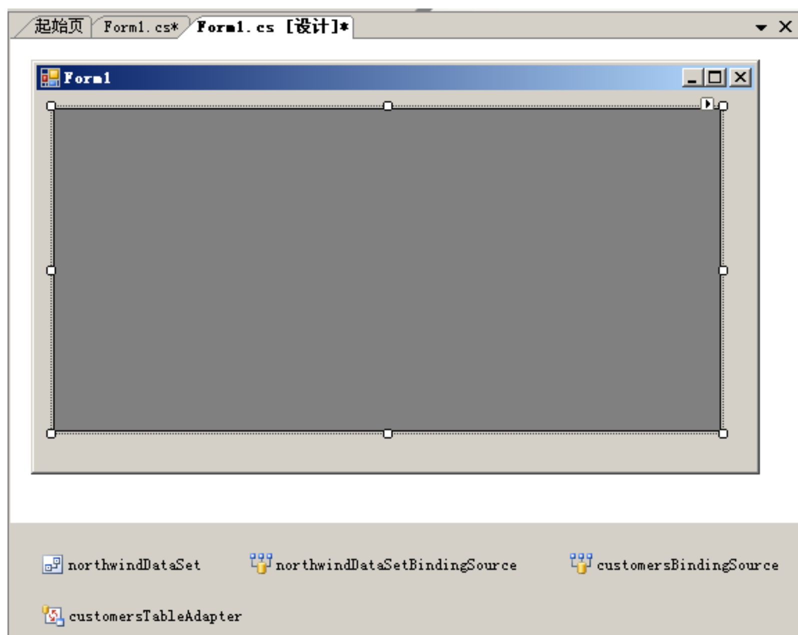


图 18-18　插入 DataGridView 控件

6）从 Visual Studio 2010“数据源”中将表 Customers 拖入到 DataGridView 控件中，并调整控件的大小，如图 18-19 所示。

7）选中 DataGridView 控件后用鼠标右键单击并选择“编辑列”选项，弹出“编辑列”对话框，如图 18-20 所示。

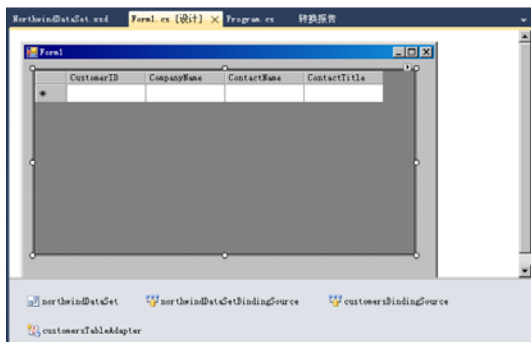


图 18-19　插入 Customers 表

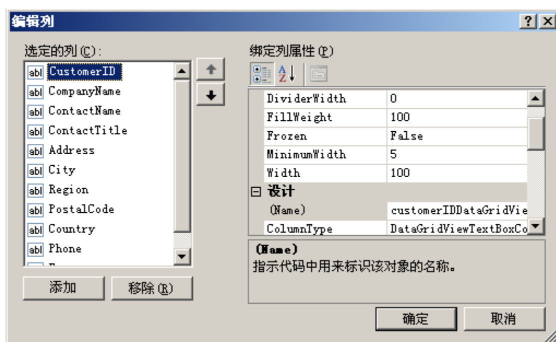


图 18-20　“编辑列”对话框

8）在“选定的列”中选择在 DataGridView 控件中显示列，在此实例只保留了前四个列值，即 CustomerID、CompanyName、ContactName 和 ContactTitle，如图 18-21 所示。

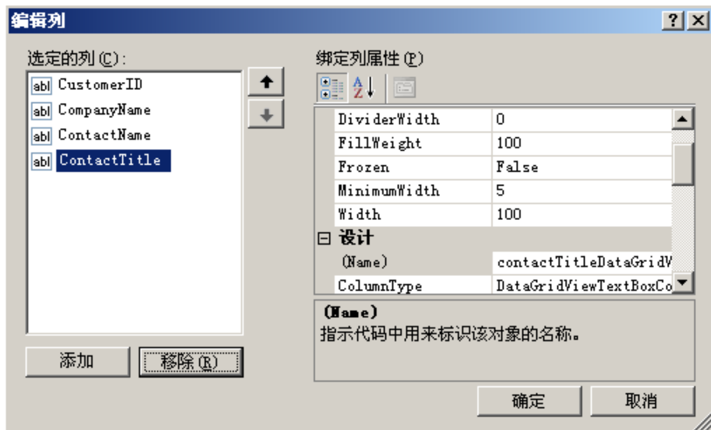


图 18-21　设置显示的列

9）单击“确定”按钮后完成整个项目实例，执行后将在窗体内显示表 Customers 中指定四个列的数据，如图 18-22 所示。

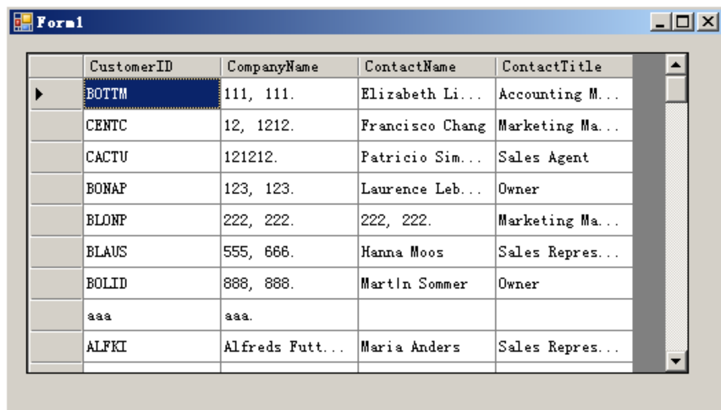


图 18-22　显示指定列数据

另外，开发人员可以设置 DataGridView 控件的各个属性，使窗体以更加精确的样式显示。下面开始简要分析上述实例的实现文件代码。

1. 数据库连接文件

数据库连接文件 app.config 的功能不言而喻，具体代码如下：

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <configSections>
    </configSections>
    <connectionStrings>
        <add name="chuli.Properties.Settings.NorthwindConnectionString"
            connectionString="Data Source=GUAN\AAA;Initial Catalog=North
wind;User ID=sa;Password=888888"
            providerName="System.Data.SqlClient" />
    </connectionStrings>
</configuration>
```

2. 窗体文件

窗体文件 Form1.cs 的功能是，定义窗体类和实例函数，分别设置载入执行事件和单击事件，主要代码如下：

```
namespace chuli
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
        private void dataGridView1_CellContentClick(object sender, Data
GridViewCellEventArgs e)
        {
        }
        private void Form1_Load(object sender, EventArgs e)
        {
            // TODO: 这行代码将数据加载到表“northwindDataSet.Customers”中，
可以根据需要移动或移除它
          this.customersTableAdapter.Fill(this.northwindDataSet.Customers);
        }
        private void dataGridView1_CellContentClick_1(object sender,
DataGridViewCellEventArgs e)
        {
        }
    }
}
```

3．控件属性设置文件

文件 Form1.Designer.cs 是一个控件属性设置文件，其功能是设置窗体内各个控件的属性，主要代码如下：

```
DataGridViewColumnHeadersHeightSizeMode.AutoSize;
          this.dataGridView1.Columns.AddRange(new System.Windows.Forms.
DataGridViewColumn[] {
          this.customerIDDataGridViewTextBoxColumn,
          this.companyNameDataGridViewTextBoxColumn,
          this.contactNameDataGridViewTextBoxColumn,
          this.contactTitleDataGridViewTextBoxColumn});
          this.dataGridView1.DataSource = this.customersBindingSource;
          this.dataGridView1.Location = new System.Drawing.Point(12,
12);
          this.dataGridView1.Name = "dataGridView1";
          this.dataGridView1.RowTemplate.Height = 23;
          this.dataGridView1.Size = new System.Drawing.Size(482, 234);
          this.dataGridView1.TabIndex = 0;
          this.dataGridView1.CellContentClick += new System.Windows.
Forms.DataGridViewCellEventHandler(this.dataGridView1_CellContentClick_
1);
          // customerIDDataGridViewTextBoxColumn
          this.customerIDDataGridViewTextBoxColumn.DataPropertyName =
"CustomerID";
          this.customerIDDataGridViewTextBoxColumn.HeaderText = "Cus
tomerID";
          this.customerIDDataGridViewTextBoxColumn.Name = "customer
IDDataGridViewTextBoxColumn";
          // contactTitleDataGridViewTextBoxColumn
          this.contactTitleDataGridViewTextBoxColumn.DataPropertyName
= "ContactTitle";
          this.contactTitleDataGridViewTextBoxColumn.HeaderText = "Con
tactTitle";
          this.contactTitleDataGridViewTextBoxColumn.Name = "contact
TitleDataGridViewTextBoxColumn";
          // Form1
          this.AutoScaleDimensions = new System.Drawing.SizeF(6F,
12F);
          this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.
Font;
          this.ClientSize = new System.Drawing.Size(518, 273);
          this.Controls.Add(this.dataGridView1);
          this.Name = "Form1";
          this.Text = "Form1";
          this.Load += new System.EventHandler(this.Form1_Load);
          ((System.ComponentModel.ISupportInitialize)(this.northwind
DataSet)).EndInit();
          ((System.ComponentModel.ISupportInitialize)(this.northwind
DataSetBindingSource)).EndInit();
          ((System.ComponentModel.ISupportInitialize)(this.customers
```

```
BindingSource)).EndInit();
            ((System.ComponentModel.ISupportInitialize)(this.dataGrid
View1)).EndInit();
            this.ResumeLayout(false);
        }
```



多学一招

除了上述基本的数据显示外，还可以设置其他属性，例如分页、数据添加、删除和修改等常见的操作处理。在此笔者简要介绍分页的基本实现方法。

1）定义几个所需的共有成员，代码如下：

```
int pageSize = 0;                  //每页显示行数
int nMax = 0;                      //总记录数
int pageCount = 0;                 //页数＝总记录数/每页显示行数
int pageCurrent = 0;               //当前页号
int nCurrent = 0;                  //当前记录行
DataSet ds = new DataSet();
DataTable dtInfo = new DataTable();
```

2）在窗体载入事件中，从数据源读取记录到 DataTable 中，代码如下：

```
//数据库连接字符串
string strConn = "SERVER=127.0.0.1;DATABASE=NORTHWIND;UID=SA;PWD=ULTR
ATEL";
SqlConnection conn = new SqlConnection(strConn);
conn.Open();
string strSql = "SELECT * FROM CUSTOMERS";
SqlDataAdapter sda = new SqlDataAdapter(strSql,conn);
sda.Fill(ds,"ds");
conn.Close();
dtInfo = ds.Tables[0];
InitDataSet();
```

3）用当前页面数据填充 DataGridView，代码如下：

```
private void InitDataSet()
       ...{
          pageSize = 20;                        //设置页面行数
          nMax = dtInfo.Rows.Count;
          pageCount=(nMax/pageSize);            //计算出总页数
          if ((nMax % pageSize) > 0) pageCount++;
          pageCurrent = 1;                      //当前页数从 1 开始
          nCurrent = 0;                         //当前记录数从 0 开始
          LoadData();
       }
private void LoadData()
       ...{
          int nStartPos = 0;                    //当前页面开始记录行
```

```
            int nEndPos = 0;                        //当前页面结束记录行
            DataTable dtTemp = dtInfo.Clone();  //克隆 DataTable 结构框架
            if (pageCurrent == pageCount)
                nEndPos = nMax;
            else
                nEndPos = pageSize * pageCurrent;
            nStartPos = nCurrent;
            lblPageCount.Text = pageCount.ToString();
            txtCurrentPage.Text = Convert.ToString(pageCurrent);
            //从元数据源复制记录行
            for (int i = nStartPos; i < nEndPos; i++)
            ...{
                dtTemp.ImportRow(dtInfo.Rows[i]);
                nCurrent++;
            }
            bdsInfo.DataSource = dtTemp;
            bdnInfo.BindingSource = bdsInfo;
            dgvInfo.DataSource = bdsInfo;
        }
```

4）编写菜单响应事件，代码如下：

```
    private void bdnInfo_ItemClicked(object sender, ToolStripItemClicked
    EventArgs e)
        ...{
            if (e.ClickedItem.Text == "关闭")
            ...{
                this.Close();
            }
            if (e.ClickedItem.Text == "上一页")
            ...{
                pageCurrent--;
                if (pageCurrent <= 0)
                ...{
                    MessageBox.Show("已经是第一页，请点击“下一页”查看！");
                    return;
                }
                else
                ...{
                    nCurrent = pageSize * (pageCurrent - 1);
                }
                LoadData();
            }
            if (e.ClickedItem.Text == "下一页")
            ...{
                pageCurrent++;
                if (pageCurrent > pageCount)
```

```
                    ...{
                       MessageBox.Show("已经是最后一页，请点击“上一页”查看！");
                    return;
                    }
                    else
                    ...{
                      nCurrent=pageSize*(pageCurrent-1);
                    }
                    LoadData();
                 }
              }
```

18.3 数据绑定

在 18.2 节中介绍了 DataGrid 控件的使用方法，DataGrid 是在运行库中唯一能够显示数据库数据的控件。在 C#开发应用中，把控件连接到数据源的过程被称为数据绑定。在 MFC 中，把数据从类变量连接到一组控件上的过程称为对话框数据交换。在本节的内容中，将向读者简要介绍数据绑定的基本知识。

18.3.1 单一绑定

支持单一绑定的控件一般一次只能显示一个值，例如文本框或单选按钮。可以使用如下代码将 DataTable（数据库）中的一列绑定到 TextBox 上：

```
DataSet ds = CreateDataSet();
textBox1.DataBindings.Add("Text", ds , "Products.ProductName");
```

在上述第一行代码中用方法 CreateDataSet()从 Products 表中检索一些数据，并保存到返回的 DataSet 中。第二行代码把控件（textBox1）的 Text 属性绑定到 Products.Product Name 列上。

上面的表 Products 是数据库 Northwind 中的表，这样文本框就显示了数据库中的一个字符串，如果需要检查这个数据是否放在正确的列上，并有正确的值，可以使用 SQL Server Query Analyze 工具，来验证 Products 表中的内容。但是上面生成的文本框，既不能滚动到下一个或上一个记录上，也不能对数据库更新。

18.3.2 数据绑定对象

在 System.Windows.Forms 命名空间中有很多相关的类，例如类 BindingContext、CurrencyManager 和 PropertyManager。当把数据绑定到窗体上的一个或多个控件上时，上述类能够相互交互。在 C#内常用的数据绑定对象有 Binding、BindingContext、BindingManagerBase、PropertyManager、CurrencyManager、BindingsCollection 和 ControlsBindingsCollection。

在 18.3.1 节中使用了 TextBox 控件的 DataBindings 属性，能够把 DataSet 的一列数据绑定到控件的 Text 属性上。属性 DataBindings 是 ControlsBindingsCollection 的一个实例：

```
textBox1.DataBindings.Add("Text", ds, "Products.ProductName");
```

通过上述代码，为 ControlBindingsCollection 添加了一个 Binding 对象。

1. BindingContext

所有 Windows 窗体都有 BindingContext 属性，此属性是在 Control 中定义的，所以几乎所有的控件都有这个属性。BindingContext 对象有一个 BindingManagerBase 实例集合，当对控件进行数据绑定时，就会创建这些实例，并把它们添加到绑定管理器对象中。

BindingContext 可以包含几个数据源，并包装在 CurrencyManager 或 PropertyManager 中。具体使用哪个类，是由数据源本身所决定的。

如果数据源包含一个项目列表，例如 DataTable 或 DataView 的对象，就使用 CurrencyManager。因为 CurrencyManager 可以在该数据源中保存当前位置。如果数据源只返回一个值，就把 PropertyManager 存储在 BindingContext 中。

只为给定的数据源创建一次 CurrencyManager 或 PropertyManager。如果把两个文本框绑定到 DataTable 的一个行上，则在 BindingContext 中只创建一个 CurrencyManager。

添加到窗体中的每个控件都链接到窗体的绑定管理器上，因此所有的控件都共享相同的实例。在最初创建一个控件时，其 BindingContext 属性为空。在把控件添加到窗体的 Controls 集合中时，就把 BindingContext 设置为该窗体的 Controls 集合。

要把控件绑定到一个列上，需要为其 DataBindings 属性添加一个入口。例如，通过下面的代码可以创建一个新的绑定：

```
textBox1.DataBindings.Add("Text", ds, "Products.ProductName");
```

在上述代码中，使用了 ControlBindingsCollection 的 Add()方法，此方法会在传递给它的参数中创建 Binding 对象的一个实例，并把它添加到绑定集合中。绑定把控件链接到数据源上，存储在 Form（或控件本身）的 BindingContext 中。数据源内部的改变会反映到控件上，控件中的改变也会反映到数据源上。

2. Binding 类

Binding 类的功能是，将控件的一个属性链接到数据源的一个成员上。当改变该成员时，控件的属性会随之更新，以反映这个改变的变化。反之亦然，如果文本框中的文本被更新，这个改变也会反映到数据源上。

可以把任何列绑定到控件的任何属性上。例如，可以把列绑定到一个文本框中，可以把另一个列绑定到文本框的颜色上，也可以把控件的属性绑定到完全不同的数据源上。

实例 89：使用数据绑定

下面将通过一个简单实例来说明数据绑定的实现流程。本实例代码保存在“光盘:\daima\17”文件夹内，项目名为 bangding。功能是通过绑定将数据显示在窗体控件内，并实现对绑定数据源的操作处理，本实例的实现步骤如下。

1）在 Visual Studio 2010 中新建一个 Windows 应用程序，如图 18-23 所示。

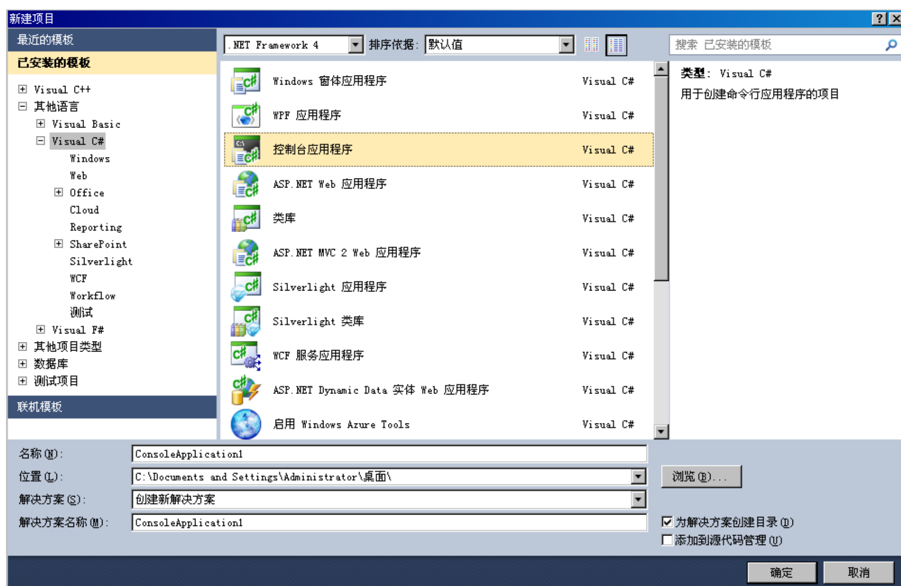


图 18-23　新建项目

2）建立和数据库 Northwind 的连接，如图 18-24 所示。

3）添加数据库 Northwind 的应用程序的数据源，如图 18-25 所示。

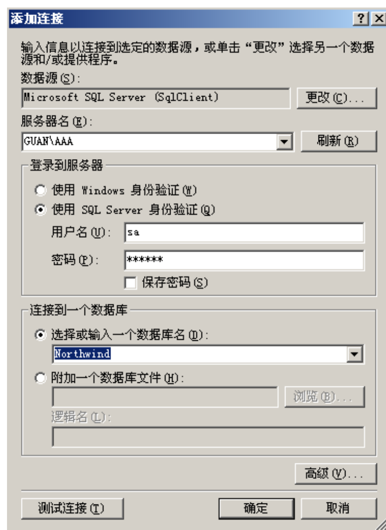


图 18-24　建立数据库连接

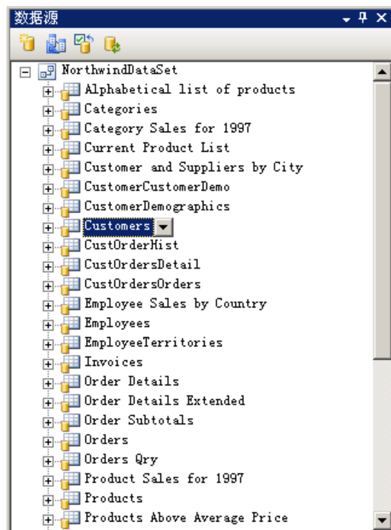


图 18-25　建立应用程序数据源

4）将“数据源”中表 Customers 的 CustomersID 列拖入到 Form1 窗体中，然后将 customerIDLabel 的 Text 属性设置为“编号 ID:”，如图 18-26 所示。

5）经过步骤 4）的操作后，控件 customerIDTextBox 的 Text 属性将自动绑定数据源 CustomersID 的值，并在属性中显示提示图标“▯”。

6）按照步骤 4）将表 Customers 的 CustomersID 列 CompanyName 和 ContactName 拖入到 Form1 窗体中，并分别设置对应的 Text 属性，如图 18-27 所示。

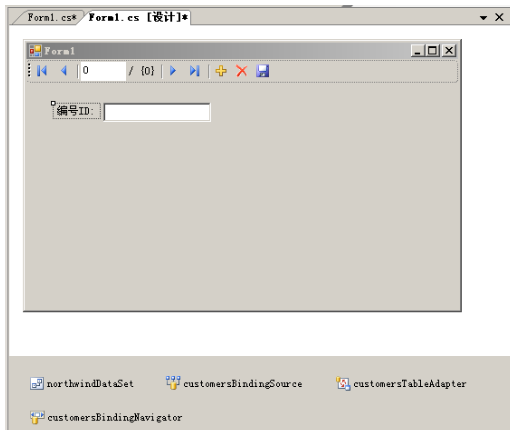


图 18-26　绑定一个列

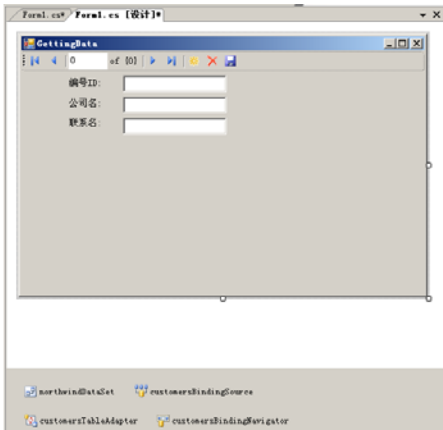


图 18-27　绑定三个列

7）经过上述操作后，会在 Form1 窗体顶部自动生成一个处理工具栏。通过工具栏里的标签可以实现对数据的操作，如图 18-28 所示。



图 18-28　自动生成的处理工具栏

8）从 Visual Studio 2010“工具箱”中拖入到 Form1 中一个 DataGridView 控件，并调整控件的大小，如图 18-29 所示。

9）从 Visual Studio 2010“数据源”中将表 Customers 拖入到 DataGridView 控件中，并只保留其中的前三列，如图 18-30 所示。

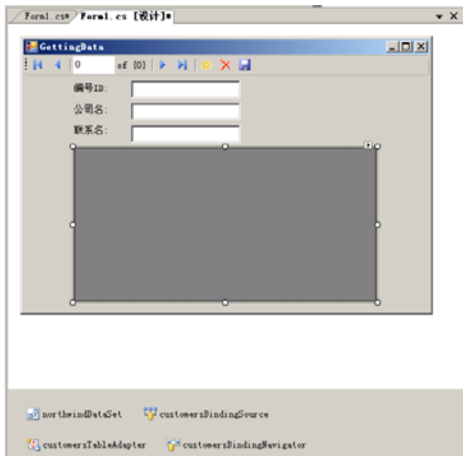


图 18-29　添加 DataGridView

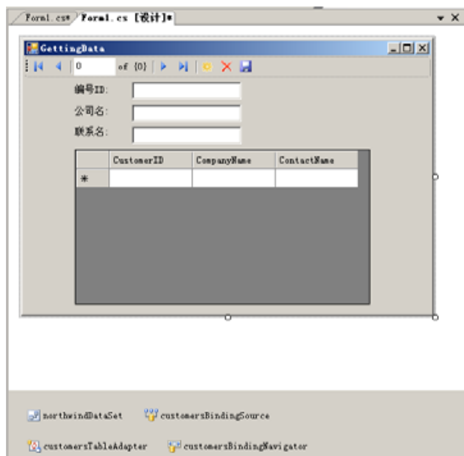


图 18-30　插入数据源数据

10）在 Visual Studio 2010 “属性”中单击 ColumnHeadersBorderStyle，设置 DataGridView 的显示样式，如图 18-31 所示。



图 18-31　设置 DataGridView 样式

11）单击“确定”按钮后完成整个项目实例，执行后将在窗体内显示对链接数据源的操作表单和工具栏标签，如图 18-32 所示。

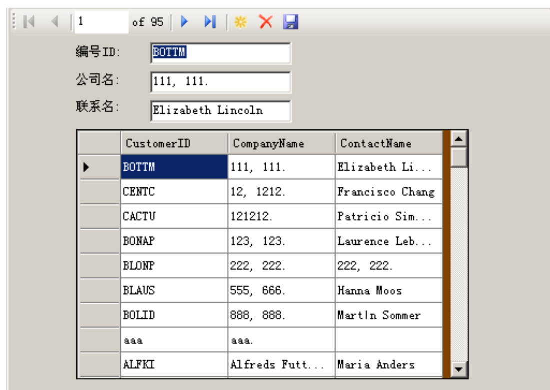


图 18-32　窗体最终执行效果

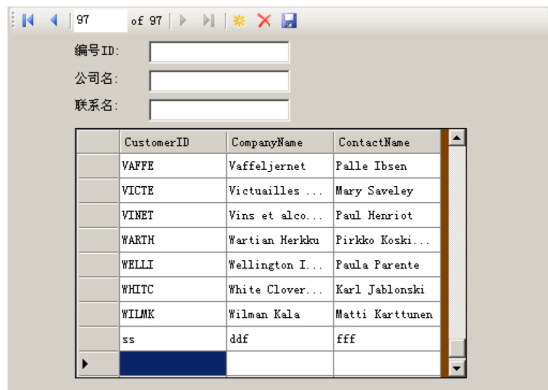


图 18-33　添加数据表单

通过单击工具栏中对应的操作图标，可以管理数据库内的数据。例如，单击添加图标则可以向库内添加新的数据，单击删除图标则删除选定的数据，单击保存图标后将更新后的数据保存，单击添加图标后将自动把上方三行表单数据清空，供用户输入新的添加数据，如图 18-33 所示。

1. 数据库连接文件

数据库连接文件 app.config 的功能不言而喻，具体代码如下：

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <configSections>
        <sectionGroup name="userSettings" type="System.Configuration.
```

```
UserSettingsGroup, System, Version=2.0.0.0, Culture =neutral, Public
KeyToken=b77a5c561934e089" >
          <section name="GettingData.Properties.Settings" type="System.
Configuration.ClientSettingsSection, System, Version=2.0.0.0, Culture=
neutral, PublicKeyToken=b77a5c561934e089" allowExeDefinition="Machine
ToLocalUser" />
       </sectionGroup>
       <sectionGroup name="applicationSettings" type="System.Configura
tion.ApplicationSettingsGroup, System, Version=2.0.0.0, Culture=neut
ral, PublicKeyToken=b77a5c561934e089" >
          <section name="GettingData.Properties.Settings" type="Sys
tem.Configuration.ClientSettingsSection, System, Version=2.0.0.0, Culture
= neutral, PublicKeyToken=b77a5c561934e089" />
       </sectionGroup>
   </configSections>
   <connectionStrings>
     <add name="GettingData.Properties.Settings.NorthwindConnectionString"
          connectionString="Data Source=GUAN\AAA;Initial Catalog=North
wind;Integrated Security=True"
          providerName="System.Data.SqlClient" />
   </connectionStrings>
   <userSettings>
       <GettingData.Properties.Settings />
   </userSettings>
   <applicationSettings>
       <GettingData.Properties.Settings />
   </applicationSettings>
</configuration>
```

2. 窗体文件

窗体文件 Form1.cs 用于定义窗体类和实例函数，并分别设置载入执行事件和单击事件，并设置工具栏图标的事件处理程序，主要代码如下：

```
namespace GettingData
{
   public partial class Form1 : Form
   {
       public Form1()
       {
          InitializeComponent();
       }
       private void bindingNavigatorSaveItem_Click(object sender, Event
Args e)
       {
          if (this.Validate())
          {
             this.customersBindingSource.EndEdit();
```

```
                this.customersTableAdapter.Update(this.northwindDataSet.
Customers);
            }
            else
            {
               System.Windows.Forms.MessageBox.Show(this, "Validation
errors occurred.", "Save", System.Windows.Forms.MessageBoxButtons.OK,
System.Windows.Forms.MessageBoxIcon.Warning);
            }
        }
      private void GettingData_Load(object sender, EventArgs e)
        {
            // TODO: This line of code loads data into the 'northwind
DataSet.Customers' table. You can move, or remove it, as needed.
            this.customersTableAdapter.Fill(this.northwindDataSet.Custo
mers);
        }
      private void customersDataGridView_CellContentClick(object sen
der, DataGridViewCellEventArgs e)
        {
        }
    }
}
```

3．控件属性设置文件

控件属性设置文件 Form1.Designer.cs 的功能是，设置窗体内各控件的属性，并设置工具栏的对应处理代码，主要实现代码如下：

```
……………………………………………………
  // 顶部绑定标单属性
            customerIDLabel.AutoSize = true;
            customerIDLabel.Location = new System.Drawing.Point(59, 34);
            customerIDLabel.Name = "customerIDLabel";
            customerIDLabel.Size = new System.Drawing.Size(47, 12);
            customerIDLabel.TabIndex = 1;
            customerIDLabel.Text = "编号 ID:";
            //
            // companyNameLabel
            //
            companyNameLabel.AutoSize = true;
             companyNameLabel.Location = new System.Drawing.Point(59,
60);
            companyNameLabel.Name = "companyNameLabel";
            companyNameLabel.Size = new System.Drawing.Size(47, 12);
            companyNameLabel.TabIndex = 3;
            companyNameLabel.Text = "公司名:";
……………………………………………………………
// 工具栏处理图标属性
```

```
            this.customersBindingNavigator.AddNewItem = this.bindingNavi
gatorAddNewItem;
            this.customersBindingNavigator.BindingSource = this.custo
mersBindingSource;
            this.customersBindingNavigator.CountItem = this.bindingNavi
gatorCountItem;
            this.customersBindingNavigator.CountItemFormat = "of {0}";
            this.customersBindingNavigator.DeleteItem = this.bindingNavi
gatorDeleteItem;
            this.customersBindingNavigator.Items.AddRange(new System.
Windows.Forms.ToolStripItem[] {
            this.bindingNavigatorMoveFirstItem,
            this.bindingNavigatorMovePreviousItem,
            this.bindingNavigatorSeparator,
            this.bindingNavigatorPositionItem,
            this.bindingNavigatorCountItem,
            this.bindingNavigatorSeparator1,
            this.bindingNavigatorMoveNextItem,
            this.bindingNavigatorMoveLastItem,
            this.bindingNavigatorSeparator2,
            this.bindingNavigatorAddNewItem,
            this.bindingNavigatorDeleteItem,
            this.bindingNavigatorSaveItem});
            this.customersBindingNavigator.Location = new System.Dra
wing.Point(0, 0);
            this.customersBindingNavigator.MoveFirstItem = this.binding
NavigatorMoveFirstItem;
            this.customersBindingNavigator.MoveLastItem = this.binding
NavigatorMoveLastItem;
            this.customersBindingNavigator.MoveNextItem = this.binding
NavigatorMoveNextItem;
           this.customersBindingNavigator.MovePreviousItem = this.binding
NavigatorMovePreviousItem;
            this.customersBindingNavigator.Name = "customersBinding Navi
gator";
            this.customersBindingNavigator.PositionItem = this.binding
NavigatorPositionItem;
            this.customersBindingNavigator.Size = new System.Drawing.
Size(505, 25);
            this.customersBindingNavigator.TabIndex = 0;
            this.customersBindingNavigator.Text = "bindingNavigator1";
           this.bindingNavigatorAddNewItem.DisplayStyle = System.Windows.
Forms.ToolStripItemDisplayStyle.Image;
            this.bindingNavigatorAddNewItem.Image = ((System.Drawing.
Image)(resources.GetObject("bindingNavigatorAddNewItem.Image")));
```

```
            this.bindingNavigatorAddNewItem.Name = "bindingNavigator
    AddNewItem";
            this.bindingNavigatorAddNewItem.Size = new System.Drawing.
    Size(23, 22);
            this.bindingNavigatorAddNewItem.Text = "Add new";
            this.bindingNavigatorCountItem.Name = "bindingNavigator
    CountItem";
            this.bindingNavigatorCountItem.Size = new System.Drawing.
    Size(41, 22);
            this.bindingNavigatorCountItem.Text = "of {0}";
            this.bindingNavigatorCountItem.ToolTipText = "Total number
    of items";

            this.bindingNavigatorDeleteItem.DisplayStyle = System.Windows.
    Forms.ToolStripItemDisplayStyle.Image;
            this.bindingNavigatorDeleteItem.Image = ((System.Drawing.
    Image)(resources.GetObject("bindingNavigatorDeleteItem.Image")));
            this.bindingNavigatorDeleteItem.Name = "bindingNavigator
    DeleteItem";
            this.bindingNavigatorDeleteItem.Size = new System.Drawing.
    Size(23, 22);
            this.bindingNavigatorDeleteItem.Text = "Delete";
            this.bindingNavigatorMoveFirstItem.DisplayStyle = System.Win
    dows.Forms.ToolStripItemDisplayStyle.Image;
            this.bindingNavigatorMoveFirstItem.Image = ((System.Drawing.
    Image)(resources.GetObject("bindingNavigatorMoveFirstItem.Image")));
            this.bindingNavigatorMoveFirstItem.Name = "bindingNavigator
    MoveFirstItem";
            this.bindingNavigatorMoveFirstItem.Size = new System.Draw
    ing.Size(23, 22);
            this.bindingNavigatorMoveFirstItem.Text = "Move first";
      ……………………………………………………………
    this.customersDataGridView.AutoGenerateColumns = false;
            this.customersDataGridView.BackgroundColor = System.
    Drawing.Color.FromArgb(((int)(((byte)(128)))), ((int)(((byte)(64)))),
    ((int)(((byte)(0)))));
            this.customersDataGridView.Columns.AddRange(new System.Windo
    ws.Forms.DataGridViewColumn[] {
            this.CustomerID,
            this.CompanyName,
            this.ContactName});
            this.customersDataGridView.DataSource = this.customers
    BindingSource;
            this.customersDataGridView.Location = new System.Drawing.
    Point(61, 112);
            this.customersDataGridView.Name = "customersDataGridView";
```

```
            this.customersDataGridView.RowTemplate.Height = 23;
            this.customersDataGridView.Size = new System.Drawing.
Size(366, 233);
            this.customersDataGridView.TabIndex = 7;
            this.customersDataGridView.CellContentClick += new System.
Windows.Forms.DataGridViewCellEventHandler(this.customersDataGridView_
CellContentClick);
```


多学一招

数据绑定是 C#项目的核心，但是有了 Visual Studio 2010 后，整个开发过程将变得十分简单。从整个实例项目的实现流程就可以看出，使用 Visual Studio 2010 可以轻松地实现对指定数据源的绑定处理。并且自动生成数据源操作事件和图标，使整个项目的开发过程变得十分轻松。并且微软还提供了大量的第三方技术支持，使开发人员可以很快上手。在此鼓励读者尽量使用 Visual Studio 2010 的集成开发环境，使自己尽快步入 Visual Studio 2010 开发和 C#开发的高手之列。

18.4　疑难问题解析

本章详细介绍了 C#在 DataGrid 和数据绑定方面的基本知识。本节将对本章中比较难以理解的问题进行讲解。

读者疑问：CurrencyManager 和 PropertyManager 很难理解，能否详细讲解？

解答：在创建 Binding 对象时，如果这是第一次绑定数据源中的数据，就会创建对应的 CurrencyManager 或 PropertyManager 对象。这个类的作用是定义当前记录在数据源中的位置，在改变当前的记录时，需要调整所有的 ListBindings。

读者疑问：在使用 DataGrid 控件显示 ADO 的数据库中的数据表内容时，一切工作都是按照标准方式进行的，从数据库的链接到数据表的生成，最后绑定数据集到 DataGrid 控件。但是按照流程却始终没有显示出数据来，而且我的五列数据最后还是变成了两列数据，并且没有内容显示出来。这是为什么？

解答：使用 DataGrid 控件默认的初始化就有两列表格，若此时在可视化的 IDE 上面对它的 Column 属性进行了编辑，那么随后的代码级的数据源绑定将失效，无法完成显示功能。解决办法就是把这个 DataGrid 控件给删除掉，然后再重新拖一个 DataGrid 进来，并且不要去修改它的 Column 属性，即可解决问题。



职场点拨——程序员创业的致命错误

比尔·盖茨和马云的神话离大家仿佛很近，一夜成名的典范有很多，不管是哪个行业，每年都会涌现出一批骄子和弄潮儿。但是作为一名程序员，如果已经决定创业，并且义无反

顾。那么一定要做好充分的准备，否则可能会功败垂成。下面是七条程序员创业的致命错误，希望能够对正在准备创业的程序员们起到一点警示作用。

1）程序员以年轻人居多，创业之初一般都有宏伟的目标，大多都喜欢大的排场。一心追求难度大的大生意，而忽视了数量多、难度小的小单子。但是到头来发现自己根本不具备做大项目的能力，并且大项目也超过了自己的经济承受能力。

2）初期创业投资者的共性是盲目追求大的投资规模，结果是造成了资产负债比率过高。如果一开始就把摊子铺得很大，忽视了种种危机的蛰伏，稍有不慎就可能爆发危机而影响全局。

3）情绪化的投资心态。都说失败乃成功之母，不去尝试就不会有收获，不去尝试，就不会积累到经验。但是不害怕失败不等于支持鲁莽，初期创业者是无法承受屡屡失败的压力的，情绪化是最可怕的投资陷阱之一。如果读者是一名即将创业的程序员，那么必须保证有清醒的头脑，冷静而客观的决策。

4）对合作缺乏真诚，频频违约。私心太重，合作缺乏诚意，不信守承诺，是投资合作中常见的事，亦常常成为投资失败的诱因。

5）思维受限制，不能立足长远，总想赚快钱，寻找短平快项目。

6）过度相信他人，不亲自进行市场调查。都说耳听为虚眼见为实，别人只是旁观者，他们看到的往往是冰山一角。正在进行的创业关乎着自己的命运，所以要从多个角度进行调研，收集完整的资料后才能做决定。

7）只求盈利，不进行创新。

所以无论是即将要创业的读者，还是正在创业的读者，笔者在此建议，一定要避免上述错误，踏上创业的成功之路。

第 19 章　GDI+图形图像编程

GDI+是 Graphics Device Interface Plus 的缩写，是公共语言运行时用来创建图形、绘制文本以及进行图形对象操作的技术。GDI+提供了在 Windows 窗体和控件上进行绘制图形、图像和文本的方法，是 C#的核心应用技术之一，通过本章能学到如下知识。

- GDI+基础。
- 图形操作。
- 测量坐标和区域。
- 字体和文本。
- 颜色操作。
- 图像操作。
- 打印操作。
- 职场点拨——谈加薪和升职。

2009 年 XX 月 XX 日，多云

今天中午吃饭时，听见同事们在讨论 A 的升职传说，据说他是本公司成立十年来第一位从程序员直接升到项目经理的。我听后很是羡慕，也在等着自己升职的那一天。



一问一答

小菜：和我同时期到公司的 A 已经升职为项目经理了，我真是羡慕。

Wisdom：呵呵，这得看你表现了，你要相信上级会公正对待的！

小菜：嗯，言归正传，本章的 GDI+很重要吗？

Wisdom：GDI+是 C#实现图形编程的基础，Windows 平台可以让开发人员不考虑特定设备的细节，就编写出磁盘读写文件程序。其中的 GDI+提供了一个抽象层，隐藏了不同视频卡之间的区别，这样就可以调用 Windows API 函数完成指定的任务。GDI+会在内部指出如何让视频卡完成要绘制的图形。还可以完成其他任务，例如能够使应用程序所使用的打印机看起来与屏幕一样，如果要打印某些东西，只需告诉系统输出的设备是打印机，再用相同的方式调用相同的 Windows API 函数即可。”

19.1　GDI+基础

GDI+是 Windows 系统显示图形图像的基础模块，其中的 Graphics 类是实现图像处理的核心。在本节的内容中，将对 GDI+的基础知识进行简要介绍。

19.1.1 GDI+

在 GDI+下，所有的绘图工作都必须通过设备环境来完成。DC 甚至可用于不涉及在屏幕或其他硬件设备上绘图的其他操作，例如在内存中修改图像。

GDI+给开发人员提供了一个相当高级的 API，但它基于旧的 Windows API，并且有 C 语言风格，所以使用起来不是很方便。GDI+在很大程度上是 GDI+和应用程序之间的一个中间层，提供了更直观、基于继承性的对象模型。

GDI+API 封装在一组托管代码中，这些类被称为 GDI+的托管类接口。GDI+基类中各个类的具体信息如下。

- System.Drawing：和基本绘图功能有关的大多数类、结构、枚举和委托。
- System.Drawing.Drawing2D：为大多数高级 2D 和矢量绘图操作提供了支持，包括消除锯齿、几何转换和图形路径。
- System.Drawing.Imaging：能够处理各种图像，例如位图、GIF 文件等。
- System.Drawing.Printing：用于把打印机或打印预览窗口作为输出设备。
- System.Drawing.Design：一些预定义的对话框、属性表和其他用户界面元素，与在设计期间扩展用户界面相关。
- System.Drawing.Text：能够对字体和字体系列执行更高级操作。

在 System.Drawing 命名空间中提供了基本的图形功能，定义了用以存储基元自身信息的类、结构、存储基元绘制方式信息的类，以及实际进行绘制的类。

19.1.2 Graphics 类

Graphics 类是 GDI+的核心，在其中提供了将对象绘制到显示设备的方法。Graphics 可以与特定设备的上下文相关联来创建图形的对象。

Graphics 类封装了绘制直线、曲线、图形、图像和文本的方法，是 GDI+实现绘制直线、曲线、图形、图像和文本的类，是 GDI+操作的基础类。GDI+绘制的实现流程如下。

1）创建 Graphics 对象。

2）通过 Graphics 对象绘制线条、形状或文本。

有如下三种方法创建 GDI+对象。

1）在窗体或控件的 Paint 事件处理方法中创建，PaintEventArgs 类参数中包含了对图形对象的引用。例如下面的代码：

```
private void Form1_Paint(object sender,System.Windows.Forms.PaintEvent
Args pe)
{
    Graphics g=pe.Graphics;
}
```

2）调用窗体或控件的 CreateGraphics 方法创建 Graphics 对象。例如下面的代码：

```
Graphics g=pe.Graphics;
g=this.CreateGraphics();
```

3）通过从 Image 继承的对象创建 Graphics 对象。例如下面的代码：

```
Bitmap mm=new Bitmap(@"D:\123.bmp");
Graphics g= Graphics.FormImage(mm);
```

创建上述 Graphics 对象后，可以用上述对象来绘制线条、形状或文本。

19.2 图形操作

这里的图形操作是指矢量图形，是用线条表示的各种形状。在本节的内容中，将详细讲解 GDI+实现图形操作的基本知识。

19.2.1 绘图工具

在 GDI+定义了一系列实现绘图处理的工具类，通过这些绘图工具可以和 Graphics 对象一起实现绘图功能。GDI+内的主要绘图工具类如下。

- ❑ Color 结构：用于设置显示不同的颜色。
- ❑ Pen 类：用于绘制线条、形状轮廓或其他几何图形。
- ❑ Brush：用于填充图形区域，例如某形状图形的颜色。

1．Color 结构

Color 结构表示 ARGB 颜色，使用 Alpha、Red、Green、和 Blue 的值来表示颜色。Color 结构以属性的方式定义了大量的有名称的颜色，例如 Color.Red、Color.PeachPuff 和 Color.Pink 等。

Color 结构内的 FormArgb()方法可以自定义生成颜色，通过设置四个值来组成指定颜色。例如，使用下面的代码创建了两种颜色：

```
Color mm = Color.FormArgb(255,0,0,255);                 //不透明蓝色
Color nn = Color.FormArgb(128,255,0,0);                 //半透明红色
```

2．Brush 类

Brush 类用于填充图形形状的内部的对象，Brush 类是一个抽象基类，不能进行实例化处理。如果需要创建一个画刷对象，则需要 Brush 类的派生类。

SolidBrush 类定义了单色画刷，画刷可以填充图形形状。例如下面的代码创建了一个实心画刷：

```
SolidBrush Brush = new SolidBrush(Color.PeachPuff);
```

通过画刷颜色的 Alpha 值来指定形状颜色和背景颜色的混合方式。例如，使用下面的代码创建了两种颜色：

```
Color mm = new SolidBrush (Color.FormArgb(255,0,0,255));     //不透明蓝
                                                             色实心画刷
Color nn = new SolidBrush (Color.FormArgb(128,255,0,0));     //半透明红
                                                             色实心画刷
```

System.Drawing.Drawing2D 命名空间中的 LinearGradientBrush 类用于定义双色渐变和自定义多色渐变画刷，所有的渐变都是沿由矩形的宽度或两个点指定的直线来定义的。LinearGradientBrush 支持水平、垂直和对角线方向的线形渐变。线形渐变中的颜色可以自行定义，使颜色非均匀的变化。例如：

```
LinearGradientBrush mm = new LinearGradientBrush(new Point(0,10)),
new Point(0,10),
new Point(200,10),
Color.FormArgb(255,255,0,0),
Color.FormArgb(255,0,0,255));
```

在上面的代码中，使用 LinearGradientBrush 的构造函数 mm 创建了一个渐变画刷。下面的代码创建了垂直的渐变画刷：

```
LinearGradientBrush mm = new LinearGradientBrush(new Point(0,10)),
new Point(10,0),
new Point(10,200),
Color.FormArgb(255,255,0,0),
Color.FormArgb(255,0,0,255));
```

3．Pen 类

Pen 类用于绘制直线或曲线对象，它通过本身的属性绘制具有指定宽度和样式的直线。可以使用 DashStyle 属性绘制虚线，DashStyle 属性值是由 DashStyle 枚举定义的，具体说明如下。

- Custom：设置用户定义的自定义画线段样式。
- Dash：设置由画线段组成的直线。
- DashDot：设置由重复的画线点图案构成的直线。
- DashDotDot：设置由重复的画线点的点图案构成的直线。
- Dot：设置由点构成的直线。
- Solid：设置直线。

看下面的一段代码：

```
Pen mm = new Pen (Color.Black,8);
Pen nn = new Pen (Color.Red);
 nn.DashStyle = System.Drawing.Drawing2D.DashStyle.Dash;
 mm.Color = Color.Pink;
 mm.Width = 4;
 mm.DashStyle = System.Drawing.Drawing2D.DashStyle.DashDot;
```

在上面的代码中，分别定义了 Pen 类函数对象 mm 和 nn，用于分别设置笔的颜色和虚实。

同样，通过画笔颜色的 Alpha 值来指定形状颜色和背景颜色的混合方式。例如，使用下面的代码创建了两种笔：

```
Pen mm = new Pen (Color.FormArgb(255,0,0,255),15);      //不透明的笔
Pen nn = new Pen (Color.FormArgb(128,0,0,255),15);      //透明的笔
```

19.2.2 绘图

使用 GDI+可以绘制如下矢量图形：直线、矩形、椭圆、弧线、多边形和基数样条。下

面通过一个具体的实例来说明使用 DrawBeziers()方法绘制曲线的过程。本实例保存在“光盘\daima\19”文件夹内，项目名为 huitu。项目实例的功能是创建一个窗体项目，然后为窗体设置一个绘图按钮事件，具体实现步骤如下。

1）在 Visual Studio 2010 中创建一个 Windows 应用程序，如图 19-1 所示。

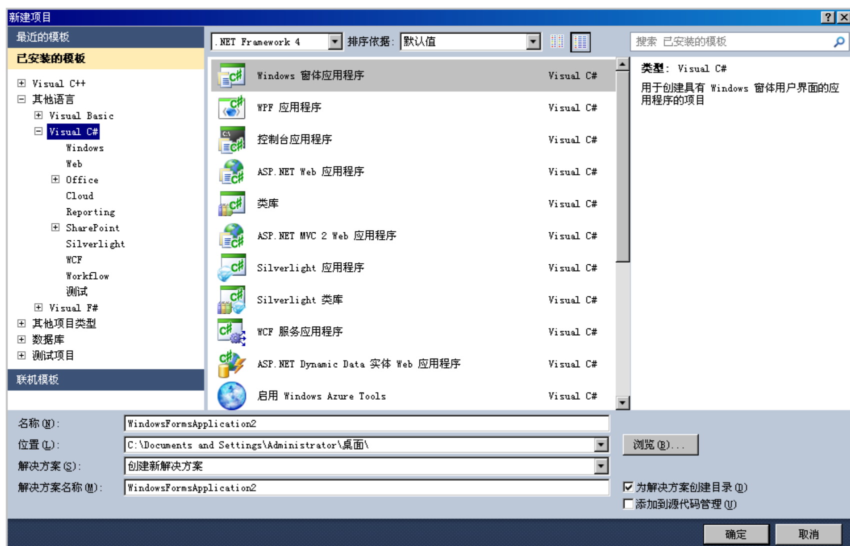


图 19-1　新建项目

2）调整窗体 Form1 的大小，从“工具箱”中拖入一个 Button 控件 button1，如图 19-2 所示。

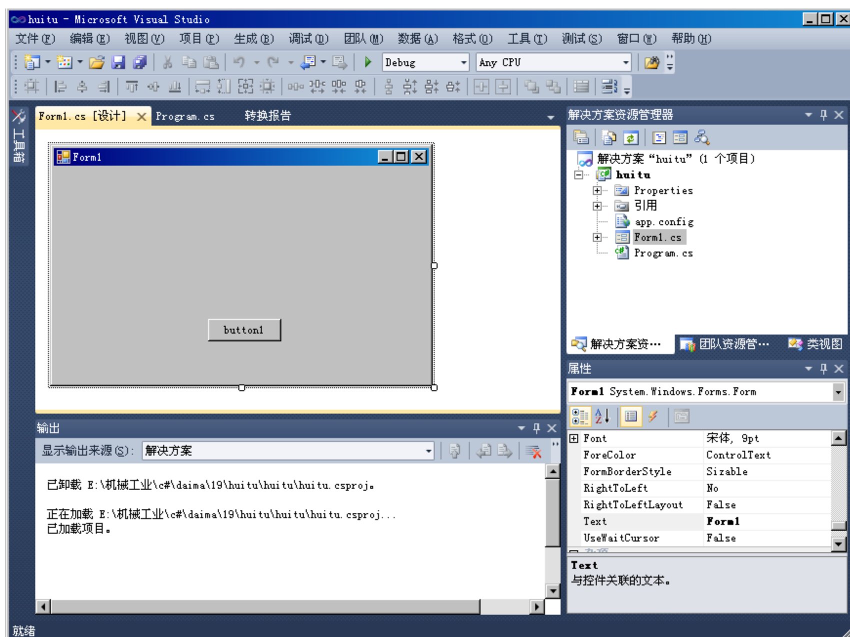


图 19-2　插入 Button 控件

3）在“属性面板”中单击 Click 事件，设置按钮单击后的事件处理程序，具体代码如下：

```
private void button1_Click(object sender, EventArgs e)
        {
            //声明一个用于绘图的 Graphics 对象
            Graphics g = this.CreateGraphics();
            //声明用于绘图的点集，程序将通过这些点的坐标进行绘图
            PointF[] points = new PointF[100];
            //分别计算各个点的函数值
            for (int i = 0; i < 100; i++)
            {
                points[i].X = i * 100;
                points[i].Y = (Compute((double)i)) * 50;
            }
            try
            {
                g.DrawBeziers(new Pen(Color.Blue), points);  //画图
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message);
            }
        }
        private float Compute(double x)
        {
            double y;
            //此处可以编写其他的函数
            y = Math.Pow(x, 2);
            return (float)y;
        }
```

4）经过上述操作后，一个简单的绘制曲线项目设置完毕。运行后将首先显示窗体效果，如图 19-3 所示。单击“button1”按钮后将在窗体内绘制一个曲线图形，如图 19-4 所示。

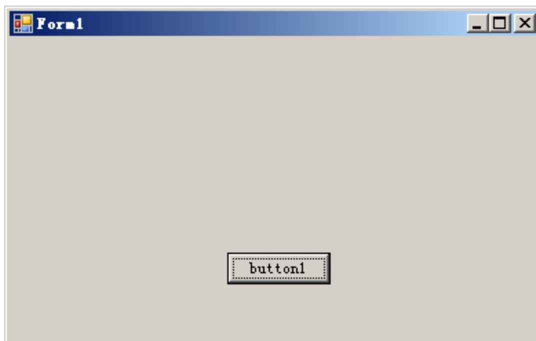


图 19-3　窗体效果



图 19-4　绘制曲线

除了上面的曲线绘制外，GDI+还能够绘制其他图形。

1．矩形和直线

使用 DrawLine()方法绘制直线，还可以使用 DrawRectangle 方法绘制矩形。

2．椭圆和弧线

使用 Graphics 类的方法 DrawEllipes()和方法 DrawArc()，可以绘制椭圆和弧线。DrawEllipes()可以绘制一个由矩形定义的椭圆，它的大多数参数和 DrawRectangle()方法的类似。而弧线是椭圆的一部分，所以 DrawArc()方法的参数也包括 DrawEllipes()方法的参数。

3．多边形

多边形是由多条线构成的闭合图形，通常使用 Graphics 类的 DrawPolygon()方法来绘制多边形。通过 DrawPolygon()方法内的 Point 参数来设置多边形的顶点，然后使用直线将各个顶点连接后就构成了多边形。

4．样条曲线

样条是一连串曲线构成的大型曲线，样条是由点的数组和张力参数指定的。通过 Graphics 类的 DrawCurve()方法可以绘制样条曲线，在绘制时需要 Pen 对象的参数和 Point 数组参数来实现。

5．实心图形

在使用上述方法进行形状绘制时，可以将图形内部进行填充。例如，使用 Brush 填充三角形或四边形的内部颜色。另外，GDI+还提供了 SolidBrush、HatchBrush、TextureBrush、LinearGradientBrush 和 PathGradientBrush 类来填充图形，上述类都是 Brush 类的派生类。

19.2.3　颜色混合

如果在创建 Brush 对象时设置的颜色属性不是 255，则此 Brush 对象绘制的实心形状颜色会和背景颜色混合。通过 Graphics 类的 CompositingMode 属性可以控制上述混合，CompositingMode 属性值是由 System.Drawing.Drawing2D.CompositingMode 枚举来定义的，该枚举的具体说明如下。

- SourceCopy：设置显示颜色时不和背景色混合。
- SourceOver：设置显示颜色时和背景色混合。

当进行图像混合处理时，Graphics 对象的 CompositingQuality 属性可以获取或设置绘制到此 Graphics 对象的合成图像的呈现质量。通过使用 CompositingQuality 枚举值可以设置符合时的质量等级，各枚举值的具体说明如下。

- AssumeLinear：假定线形值。
- Default：默认质量。
- GammaCorrected：用灰度校正。
- HighQuality：高质量、低速度符合。
- HighSpeed：高速度、低质量符合。
- Invalid：无效质量。

实例 90：实现颜色混合处理

下面通过一个具体实例来说明颜色混合处理的过程。本实例保存在“光盘:\daima\19”文件夹内，项目名为 WindowsApplication1。功能是创建一个窗体项目并在窗体内显示三原色图形。具体实现步骤如下。

1）在 Visual Studio 2010 中创建一个 Windows 应用程序，如图 19−5 所示。

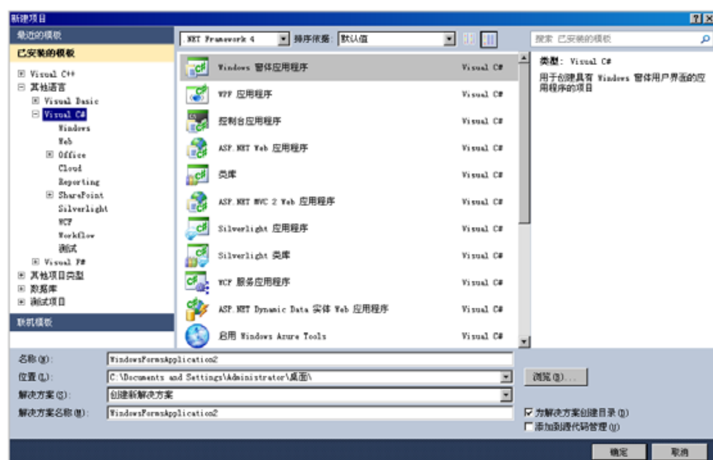


图 19-5　新建项目

2）调整窗体 Form1 的大小，然后编写绘图处理代码，主要代码如下：

```
namespace WindowsApplication1
{
    public partial class Form1 : Form
    {
    // 设置窗体属性
        public Form1()
    {
        this.Text = "三原色";
        this.StartPosition = FormStartPosition.CenterScreen;
        this.FormBorderStyle = FormBorderStyle.FixedDialog;
        this.MaximizeBox = false;
        this.MinimizeBox = false;
        this.Width = 400;
        this.Height = 430;
    }
    // 绘制三原色图形
    protected override void OnPaint(PaintEventArgs e)
    {
        // 获取 Graphics 对象
        Graphics g = e.Graphics;
        g.CompositingQuality = System.Drawing.Drawing2D.CompositingQua
lity.GammaCorrected;
        int l = (int)(e.ClipRectangle.Width * 0.4);
        int x = (int)(e.ClipRectangle.Width * 0.3);
        int y = (int)(e.ClipRectangle.Width * 0.2);
        Rectangle aa = new Rectangle(x, y, l, l);
        Rectangle bb = new Rectangle(y, l, l, l);
        Rectangle cc = new Rectangle(l, l, l, l);
        SolidBrush dd = new SolidBrush(Color.Black);
```

```
            SolidBrush redBrush = new SolidBrush(Color.FromArgb(128, 255, 0, 0));
            SolidBrush greenBrush = new SolidBrush(Color.FromArgb(128, 0, 255, 0));
            SolidBrush blueBrush = new SolidBrush(Color.FromArgb(128, 0, 0, 255));
            g.FillRectangle(dd, e.ClipRectangle);
            g.FillEllipse(redBrush, aa);
            g.FillEllipse(greenBrush, bb);
            g.FillEllipse(blueBrush, cc);
            // 释放画刷
            dd.Dispose();
            redBrush.Dispose();
            greenBrush.Dispose();
            blueBrush.Dispose();
        }
            private void Form1_Load(object sender, EventArgs e)
            {

            }
    }
    }
```

上述代码的具体实现流程如下。

- ❑ 定义窗体构造函数 Form1，设置窗体的显示属性。
- ❑ 定义函数 OnPaint，绘制三原色图形。
- ❑ 通过 SolidBrush 分别创建五个不同颜色的画刷，各画刷的具体说明如下。
- ● dd：黑色实心不透明画刷。
- ● redBrush：红色实心半透明画刷。
- ● greenBrush：绿色实心半透明画刷。
- ● blueBrush：蓝色实心半透明画刷。
- ❑ 通过 FillRectangle()方法分别绘制红色、绿色和蓝色的圆。
- ❑ 使用 Dispose()释放画刷。

执行后将在窗体内显示一个三原色图形，如图 19-6 所示。

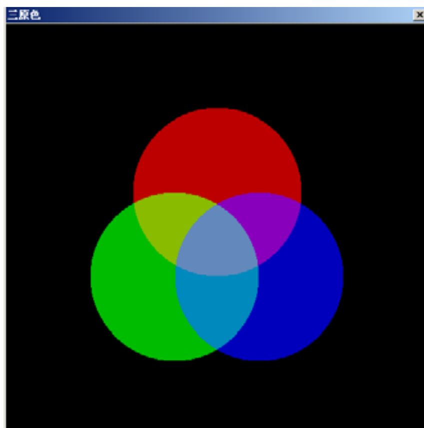


图 19-6　三原色图形

19.3 测量坐标和区域

在本书前面的内容中，了解到了用基本结构 Rectangle 可以表示矩形的坐标。GDI+使用几个类似的结构来表示坐标或区域，它们都是在 System.Drawing 命名空间中定义的，它们的结构类型的具体使用方法基本类似，具体信息如下。

- struct Point 结构：主要属性有 x 和 y。
- struct PointF 结构：主要属性有 x 和 y。
- struct Size 和 struct SizeF 结构：主要属性有 Width 和 Height。
- struct Rectangle 和 struct RectangleF 结构：主要属性有 Left，Right ，Top，Bottom，Width，Height，X，Y，Location，Size

19.3.1 Point 和 PointF

GDI+中结构最简单的是 Point，在数学上它完全等价于一个二维矢量，包含两个公共整型属性，表示它与某个特定位置的水平和垂直距离。

例如，为了从点 A 到点 B，需要水平移动 20 个单位，并向下垂直移动 10 个单位。在坐标图中使用 x 和 y，可以通过如下代码创建一个 Point 结构来表示：

```
Point ab = new Point(20, 10);
Console.WriteLine("Moved {0} across, {1} down", ab.X, ab.Y);
```

其中，属性 X 和 Y 都是可读写属性，可以在 Point 中直接设置这些值。例如下面的代码：

```
Point ab = new Point();
ab.X = 20;
ab.Y = 10;
Console.WriteLine("Moved {0} across, {1} down", ab.X, ab.Y);
```



按照惯例，水平和垂直坐标表示为小写的 x 和 y，但是对应的 Point 属性是大写 X 和 Y。这是因为在 C#中，公共属性的一般约定是名称以大写字母开头。

PointF 与 Point 完全相同，但 X 和 Y 属性的类型是 float，而不是 int。PointF 用于坐标不是整数值的情况。已经为这些结构定义了数据类型转换，这样就可以把 Point 隐式转换为 PointF。但是上述转换没有相应的逆过程，要把 PointF 转换为 Point，必须显式地复制值，或使用下面的三个转换方法 Round()、Truncate()和 Ceiling()：

```
PointF mm = new PointF(20.5F, 10.9F);
Point nn = new Point();
nn.X = (int)mm.X;
nn.Y = (int)mm.Y;
Point ab1 = Point.Roundmm);
Point ab2 = Point.Truncate(mm);
Point ab3 = Point.Ceiling(mm);
```

```
PointF mm = nn;
```

在默认情况下，GDI+是以电脑屏幕上的像素为单位，Graphics 对象方法把它们接收到的坐标看作其参数的方式。例如，点 new Point（20,10）表示在屏幕上水平移动 20 个像素，向下垂直移动 10 个像素。通常这些像素从窗口客户区域的左上角开始测量，如上面的示例所示。但是，情况并不总是如此。例如，在某些情况下，需要以窗口的左上角作为原点来绘图，甚至以屏幕的左上角为原点。但在大多数情况下，除非文档说明书说明，否则都可以假定像素值是相对于客户区域的左上角。

19.3.2　Size 和 SizeF

与前面介绍的 Point 和 PointF 一样，Size 也有两个变体。Size 和 SizeF 的区别如下。

- Size 结构用于 int 类型。
- SizeF 用于 float 类型。

在大多数情况下，Size 结构与 Point 结构是相同的，也有两个整型属性，分别表示水平和垂直距离。但上述两个属性的名称不是 X 和 Y，而是 Width 和 Height。例如，为了从点 A 到点 B，需要水平移动 20 个单位，并向下垂直移动 10 个单位。在坐标图中使用 Width 和 Height，可以通过如下代码创建一个 Size 结构来表示：

```
Size ab = new Size(20,10);
Console.WriteLine("Moved {0} across, {1} down", ab.Width, ab.Height);
```

Size 结构在数学上与 Point 结构表示的含义相同，但在概念上它使用的方式略有不同。Point 用于说明实体在什么地方，而 Size 用于说明实体有多大。但是，Size 和 Point 是紧密相关的，两者之间可以进行显式转换。例如下面的代码：

```
Point point = new Point(20, 10);
Size size = (Size) point;
Point anotherPoint = (Point) size;
```

如果有一个矩形，其左上角的坐标是（0,0），大小是（50,50）。这个矩形的大小是（50,50），可以用一个 Size 实例来表示。其右下角的坐标也是（50,50），但它由一个 Point 实例来表示。要理解这个区别，假定在另一个位置绘制该矩形，其左上角的坐标是（10,10）：

```
dc.DrawRectangle(bluePen, 10,10,50,50);
```

现在其右下角的坐标是（60,60），但大小不变，仍是（50,50）。

Point 和 Size 结构的相加运算符都已经重载了，所以可以把一个 Size 加到 Point 结构上得到另一个 Point 结构。例如下面的代码：

```
static void Main(string[] args)
{
Point topLeft = new Point(10,10);
Size rectangleSize = new Size(50,50);
Point bottomRight = topLeft + rectangleSize;
```

```
        Console.WriteLine("topLeft = " + topLeft);
        Console.WriteLine("bottomRight = " + bottomRight);
        Console.WriteLine("Size = " + rectangleSize);
        }
```

也可以从一个 Point 减去某个 Size，得到另一个 Point。并且也可以把两个 Size 加在一起，得到另一个 Size。但是不能把一个 Point 加到另一个 Point 上，不支持＋运算符的任何重载进行这样的操作。

可以在 Point 和 Size 之间实现显式的数据类型转换，例如下面的代码：

```
        Point topLeft = new Point(10,10);
        Size s1 = (Size)topLeft;
        Point p1 = (Point)s1;
```

在上述数据类型转换时，s1.Width 被赋予 topLeft.X 的值，s1.Height 被赋予 topLeft.Y 的值，因此 s1 包含(10,10)。p1 最终的值与 topLeft 的值相同。

19.3.3　Rectangle 和 RectangleF

结构 Rectangle 和 RectangleF 能够表示屏幕上的一个矩形区域，两者的使用方法基本类似。只是 RectangleF 的属性类型是 float 类型，而 Rectangle 的属性类型是 int 类型。

可以将 Rectangle 看做是一个 Point 和一个 Size 组成的结构，其中 Point 表示矩形的左上角，Size 表示其大小。Rectangle 的构造函数可以把 Point 和 Size 作为其参数，然后即刻绘制一个矩形。例如下面的代码：

```
        Graphics aa = e.Graphics;
        Pen bb = new Pen(Color.Blue, 3);
        Point cc = new Point(0,0);
        Size dd = new Size(50,50);
        Rectangle tu = new Rectangle(cc, dd);
        dc.DrawRectangle(bb, tu);
```

另外，上述代码也可以使用 Graphics.DrawRectangle()的另一个重载方法，使用 Pen 和 Rectangle 结构作为结构参数。

通过按顺序提供矩形的左上角水平和垂直坐标，宽度和高度（它们都是数字），可以构造一个 Rectangle，例如下面的代码：

```
        Rectangle rectangleArea = new Rectangle(0, 0, 50, 50);
```

Rectangle 使用本身的读写属性来组合设置它的维数，各属性的具体说明如下。

- 属性 int Left：设置左边界的 x 坐标。
- 属性 int Right：设置右边界的 x 坐标。
- 属性 int Top：设置顶边的 y 坐标。
- 属性 int Bottom：设置底边的 y 坐标。
- 属性 int X：设置左边界的 x 坐标。

- 属性 int Y：设置顶边的 y 坐标。
- 属性 int Width：设置矩形的宽度。
- 属性 int Height：设置矩形的高度。
- 属性 Point Location：设置左上角原点。
- 属性 Size Size：设置矩形的大小。

上述属性都不是独立的，需要相互结合使用才能定义出一个图形。

19.4　字体和文本

在 GDI+绘图处理过程中，可以使用机器上已安装的任何字体。在本节的内容中，将对 GDI+zoti 和文本处理的基本信息进行简要介绍。

19.4.1　字体处理

字体指定了文本的显示格式，例如文本样式和字体大小等。在 GDI+模型中，将样式相同但字型不同的字体划分为不同的字体系列，具体说明如下。

- 常规系列：即 Regular。
- 粗体系列：即 Bold。
- 斜体系列：即 Italic。
- 粗斜体系列：即 Bold Italic。

GDI+进行文本绘制时，需要首先构造一个 FontFamiliy 对象，然后使用它来构造一个 Font 对象。FontFamiliy 对象用于指定字体的样式，Font 对象用于指定字号、字形和单位。

FontFamiliy 类定义了一组有相似设计但在形势上有差异的字样。FontFamiliy 类的 Familiies 属性返回一个数组，数组包含了与当前图形上下文相关的所有 FontFamiliy 对象。看下面的一段代码：

```
namespace WindowsApplication2
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
        private void Form1_Load(object sender, EventArgs e)
        {
            string mm = "";
            foreach (FontFamily aa in FontFamily.Families)
            {
                mm += aa.Name + ",";
            }
            MessageBox.Show(mm,"可以使用");
        }
```

```
        }
    }
```

在上述代码中，首先定义了 string 类型变量 mm，用于保存当前项目能够使用的字体系列。然后使用 foreach 逐一将 Families 数组内的字体读取出来，最后使用对话框输出字体数据。执行后的运行结果如图 19-7 所示。

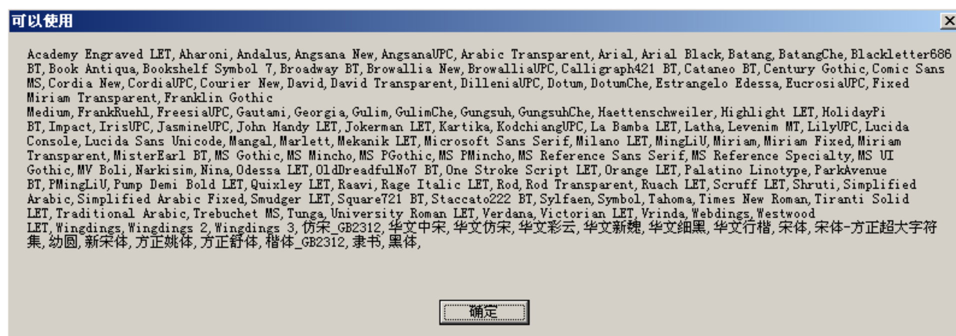


图 19-7　对话框显示字体

Font 类通过 Style 属性设置字体的样式信息，其值是由 FontStyle 枚举定义的。FontStyle 枚举值的具体说明如下。

- Bold：加粗文本。
- Italic：斜体文本。
- Regular：普通文本。
- Strikeout：删除线文本。
- Underline：下划线文本。

Font 类通过 Unit 属性设置字体的大小单位，其值是由 GraphicsUnit 枚举定义的。GraphicsUnit 枚举值的具体说明如下。

- Display：设置显示设备的大小单位，例如显示器的单位是像素，打印机的单位是 1/100 英寸。
- Document：使用文档单位作为大小单位，即 1/300 英寸。
- Inch：使用英寸。
- Pixel：使用设备像素。
- Millimeter：使用毫米。
- Point：使用打印机点作为单位，即 1/72 英寸。
- Word：使用世界坐标系单位。

例如，下面的代码构造了一个大小为 16 像素、显示为粗体的 Arial 字体。

```
FontFamiliy mm = new FontFamiliy("Arial");
Font nn = new Font(
mm,
16,
FontStyle.Bold,
```

```
GraphicsUnit.Pixel
)
```

在.NET 框架中，使用通用对话框控件 FontDialog 来选取字体和字体的属性。在 FontDialog 对话框中选取字体后，将从 Font 属性返回选取的字体。看下面的一段代码：

```
public Font GetFontFromFontDialog()
       {
         FontDialog mm = new FontDialog();
        if (FontDialog.ShowDialog()==DialogResult.OK)
        return mm.Font;
       }
   return null;
   }
```

在上面的代码中，将输出一个 FontDialog 对话框，用户选择一种字体并单击“OK”按钮后，将返回所选择的字体。

19.4.2 文本处理

在 C#中，可以使用 Graphics 类的 DrawString()方法、System.Windows.Forms.TextRender 类的 DrawText()方法来绘制文本。如果支持 Point、PointF 对象或（x,y）坐标作为参数来指定某点的位置，上述两个方法可以从指定的点开始在水平方向上绘制文本。如果接受 Rectangle 或 RectangleF 对象作为参数来设置一个矩形，上述两个方法可以在指定的矩形区域内绘制文本，并且都接受 Font 对象来确定显示文本的字体。DrawSting()方法使用 Brush 来设置文本的颜色和填充方式，DrawText()方法使用 Color 来设置文本的颜色。看下面的一段代码：

```
protected override void OnPaint(PaintEventArgs e)
       {
          Graphics g = e.Graphics;
          LinearGradientBrush ll = new LinearGradientBrush(new Point
(0,0),new Point(450,450), Color.Red,Color.Blue);
       }
  using (Font font = new Font("SimSum",48,FontStyle.Bold,Graphics
Unit.Pixel))
       {
          Point aa=new Point(10,10);
          g.DrawSting("dddddd",font,ll,aa);
          Rectangle bb=new Rectangle(10,80,500,200);
          g.DrawSting("dddddd",font,ll,bb);
       }
```

在上述代码中，创建了一个水平方向的线形渐变画刷。

当使用 DrawString()方法和 DrawText()方法绘制文本时，可以设置文本的绘制格式，例如行距、对齐方式和书写方向等。DrawString()方法接受 StringFormat 对象，而 StringFormat

对象是通过 StringFormatFlags 枚举的成员构造的。DrawText()方法接受 TextFormatFlags 枚举的常数。上述两个枚举都可以设置文本字符串的显示和布局信息，都有一个共同的 FlagsAttribute 属性，允许成员值按位进行组合。两枚举定义的常数也很类似，StringFormatFlags 枚举的成员的具体信息如表 19－1 所示。

表 19－1　StringFormatFlags 枚举的成员信息

成　　员	说　　明
Bottom	对齐边框的底部文本
Default	默认左对齐
EndEllipsis	删除修正行结尾，以省略号换行
ExpandTabs	扩展制表符
ExternalLeading	在行高中包括字体的外部间隙
GlyphOverhangPadding	在边框中添加空白边距，以容纳延伸的标志字符
HidePrefix	仅用于 Windows XP 和 Windows 2000 系统
HorizontalCenter	忽略文本中的&符号
Internal	比较 NoPrefix 和 PrefixOnly
Left	将边框文本水平居中对齐
LeftAndRightPadding	使用系统字体计算文本规格
ModifyString	对齐剪辑区域的左侧文本
NoClipping	边框两边添加空白边框
NoFullWidthCharacterBreak	修改指定字符串，使其与显示文本匹配
NoPadding	允许标志符的延伸部分和延伸到格式化矩形外的未换行文本
NoPrefix	仅用于 Windows 98、Windows Me、Windows 2000 和 Windows XP 系统
PathEllipsis	消除在双倍宽度字符串中的换行符
PrefixOnly	不在边框添加空白边距
PreserveGraphicsClipping	关闭对前缀字符的处理
PreserveGraphicsTranslateTransform	比较 HidePrefix 和 PrefixOnly
Right	删除修整行的中心并使用省略号进行替换
RightToLeft	仅用于 Windows XP 和 Windows 2000 系统
SingleLine	在&前缀字符后的字符位置绘制下画线，但是不在其他位置绘制
TextBoxControl	比较 HidePrefix 和 NoPrefix
Top	保留 Graphics 指定的剪辑
VerticalCenter	保留 Graphics 指定的转换
WordBreak	对齐剪辑区域右侧的文本
WordEllipsis	设置从右向左显示文本

19.5　颜色操作

在 GDI+中，使用 System.Drawing.Color 结构的实例来表示颜色。在大多数情况下，Color 结构被初始化后就不能使用对应的 Color 实例对该结构进行操作了，而只能把它传送

给其他需要 Color 的调用方法。

19.5.1　红绿蓝值

红绿蓝值即平面设计中的 RGB 值。在程序开发中需要对这些颜色进行索引，才能指定在给定的某个像素上要显示什么颜色。给颜色进行索引的最常见方式是对它们进行红绿蓝的成份（component）分配。

人眼可以分辨的任何颜色都是由一定量的红色光、绿色光和蓝色光组成的。这些光称为成份。如果每种成份的光分为 256 种不同的强度，并且提供了足够平滑的过渡，就可以把人眼能分辨出来的图像显示为和照片一样的质量。所以在设置颜色时，可以给出这些成份的具体量，值在 0～255 之间，其中 0 表示没有这种成份，255 表示这种成份的光达到最大的强度。

GDI+可以调用静态函数 Color.FromArgb()来指定该颜色的红绿蓝值。微软没有为此提供专用的构造函数，原因是除了一般的 RGB 成分外，还有其他方式可以指定构造函数。

19.5.2　命名的颜色

在 C#中，通常使用 FromArgb 实现颜色构造，因为它可以指定人眼能辨识出的任何颜色。如果要得到一种标准、众所周知的纯色，例如红色或蓝色，这是比较简单的。因为微软在 Color 中提供了许多静态属性，每个属性都返回一种命名的颜色。例如在下面的代码中，把窗口的背景色设置为白色时使用了静态属性：

```
this.BackColor = Color.White;
```

上述代码和下面的代码是等效的：

```
this.BackColor = Color.FromArgb(255, 255 , 255);
```

在颜色定义中有几百种这样的静态颜色，具体信息读者可以参阅 MSDN 的相关文档说明。

19.6　图像操作

在 GDI+中，使用抽象基类 Image 来处理位图和矢量图。位图图像的 Bitmap 类和矢量图形 Metafile 类都继承于 Image，而 Bitmap 类扩展了 Image 类的功能，实现了加载、保存等功能。Metafile 类也扩展了 Image 类的功能，实现了用于记录和检查矢量图形的更多方法。

19.6.1　位图文件简介

在类 System.Drawing.Bitmap 中封装了 GDI+位图，它可以处理由像素数据定义的图像对象。Bitmap 类支持多种文件格式，例如常见的 BMP、GIF、JPEG 和 PNG 等。Bitmap 类可以使用多种形式的构造函数进行重载，可以通过图像文件名和图像文件流等来构造图像，也可以创建一个空图像并在其中进行绘制。

Bitmap 类封装了图像的各种属性和方法，通过这些属性和方法可以实现和图像有关的各种操作处理。Bitmap 类的常用属性信息如表 19-2 所示。

表 19-2　Bitmap 类的常用属性列表

属　　性	说　　明
Flags	获取图像像素数据的属性标志
FrameDimensionsList	获取 GUID 数组
Height	获取图像的高度
HorizontalResolution	获取图像的水平分辨率
Palette	获取图像的调色板
PhysicalDimension	获取图像的宽度和高度
PixelFormat	获取图像的像素格式
PropertyIdList	获取图像中的属性 ID
PropertyItems	获取图像中的所有属性
RawFormat	获取图像中的格式
Size	获取图像中的像素单位大小
Tag	获取图像中的附加数据对象
VerticalResolution	获取图像中的垂直分辨率
Width	获取图像中的宽度

Bitmap 类的常用方法信息如表 19-3 所示。

表 19-3　Bitmap 类的常用方法列表

方　　法	说　　明
Dispose()	释放图像资源
FromFile()	从指定文件创建图像
FromHbitmap()	从 Windows 句柄创建 Bitmap
FromHicon()	从图标的 Windows 句柄创建 Bitmap
FromResource()	从指定的 Windows 资源创建 Bitmap
FromStream()	从指定的数据流创建图像
GetBounds()	以指定单位获取图像界限
GetEncoderParameterList()	返回图像编码器支持的参数信息
GetFrameCount()	返回指定纬度的帧数
GetHbitmap()	从 GDI+Bitmap 创建 GDI 位图对象
GetPixel()	获取 Bitmap 中指定像素的颜色
GetPixelFormatSize()	返回指定像素格式的颜色深度
GetPropertyItem()	获取图像中的某属性
GetThumbnailImage()	返回图像的缩略图
IsAlphaPixelFormat()	返回此像素格式是否包含 alpha 信息
IsCanonicalPixelFormat()	返回此像素格式是否为 32 位
IsExtendedPixelFormat()	返回此像素格式是否为 64 位
LockBits()	将 Bitmap 锁定到内存
MakeTransparent()	使用默认透明色
RemovePropertyItem()	从属性中删除某属性
RotateFlip()	旋转、翻转或同时旋转和翻转图像
Save()	将图像保存
SaveAdd()	将信息添加到图像中
SelectActiveFrame()	选择由纬度和索引指定的一帧
SetPixel()	获取 Bitmap 指定的像素颜色
SetPropertyItem()	在图像中存储一个属性
SetResolution()	设置 Bitmap 的分辨率
UnlockBits()	内存中解锁 Bitmap

在开发中常用的是方法 FromFile()和方法 Save()，分别用于创建和保存图像。

19.6.2　矢量图文件简介

在矢量图文件中存储了一系列的绘图命令和设置，用 Metafile 对象记录的命令和设置可以存储在内存中，也可以保存在文件或流中。GDI+主要支持如下三种矢量图文件格式。

1）WMF：Windows 矢量图形文件。

2）EMF：增强的矢量图形文件。

3）EMF+：EMF 的扩展，可以存储 GDI+记录。

类 System.Drawing.Metafile 用于定义图形的图元文件，它包含了一系列图形操作的记录，这些操作可以被构造和显示。通过 Metafile 类，可以使用多种形式的构造函数进行重载，通过图形对象和字符串对象来构造 Metafile 对象，也可构造空的 Metafile 对象。

Metafile 类封装了矢量图形的各种属性和方法，通过这些属性和方法可以实现和图像有关的各种操作处理。Metafile 类的常用属性信息如表 19－4 所示。

表 19－4　Metafile 类的常用属性列表

属　　性	说　　明
Flags	获取图像像素数据的属性标志
FrameDimensionsList	获取 GUID 数组
Height	获取图像的高度
HorizontalResolution	获取图像的水平分辨率
Palette	获取图像的调色板
PhysicalDimension	获取图像的宽度和高度
PixelFormat	获取图像的像素格式
PropertyIdList	获取图像中的属性 ID
PropertyItems	获取图像中的所有属性
RawFormat	获取图像中的格式
Size	获取图像中的像素单位大小
Tag	获取图像中的附加数据对象
VerticalResolution	获取图像中的垂直分辨率
Width	获取图像中的宽度

Metafile 类的常用方法信息如表 19–5 所示。

表 19–5　Metafile 类的常用方法列表

方　　法	说　　明
Dispose()	释放图像资源
FromFile()	从指定文件创建图像
FromHbitmap()	从 Windows 句柄创建 Bitmap
FromHicon()	从图标的 Windows 句柄创建 Bitmap
FromStream()	从指定的数据流创建图像

（续）

方　法	说　明
GetBounds()	以指定单位获取图像界限
GetEncoderParameterList()	返回图像编码器支持的参数信息
GetFrameCount()	返回指定纬度的帧数
GetHenhumetafile()	返回增强型 Metafile 的 Windows 句柄
GetMetafileHeader()	返回和 Metafile 关联的 GetMetafileHeader
GetPixelFormatSize()	返回指定像素格式的颜色深度
GetPropertyItem()	获取图像中的某属性
GetThumbnailImage()	返回图像的缩略图
IsAlphaPixelFormat()	返回此像素格式是否包含 Alpha 信息
IsCanonicalPixelFormat()	返回此像素格式是否为 32 位
IsExtendedPixelFormat()	返回此像素格式是否为 64 位
PlayRecord()	播放单个矢量图形文件记录
MakeTransparent()	使用默认透明色
RemovePropertyItem()	从属性中删除某属性
RotateFlip()	旋转、翻转或同时旋转和翻转图像
Save()	将图像保存
SaveAdd()	将信息添加到图像中
SelectActiveFrame()	选择由纬度和索引指定的一帧
SetPropertyItem()	在图像中存储一个属性

19.6.3　GDI+图像处理

在 Graphics 对象中，可以使用 DrawImage 方法绘制从任何 Image 类派生来的图像，另外也可以使用 PictureBox 控件来显示常见格式的图像。使用 PictureBox 控件的属性可以控制图像的显示格式，其属性是由 PictureBoxSizeMode 枚举定义的。PictureBoxSizeMode 各个枚举值的具体说明如下。

- ❑ Normal：将图像置于 PictureBox 的左上角。
- ❑ StretchImage：将图像拉伸，并调整为控件的大小。
- ❑ AutoSize：将控件大小设置为图像的大小。
- ❑ CenterImage：使图像居于工作区中心。
- ❑ Zoom：图像将按其原来大小比例增加或减小。

实例 91：使用 PictureBox 控件

下面通过一个具体的实例来说明 PictureBox 控件的具体使用过程。本实例保存在“光盘:\daima\19”文件夹内，项目名为 tupian。功能是创建一个窗体项目，在窗体内实现图片的打开、保存和不同视图效果的功能。具体实现步骤如下。

1）在 Visual Studio 2010 中创建一个 Windows 应用程序，如图 19-8 所示。

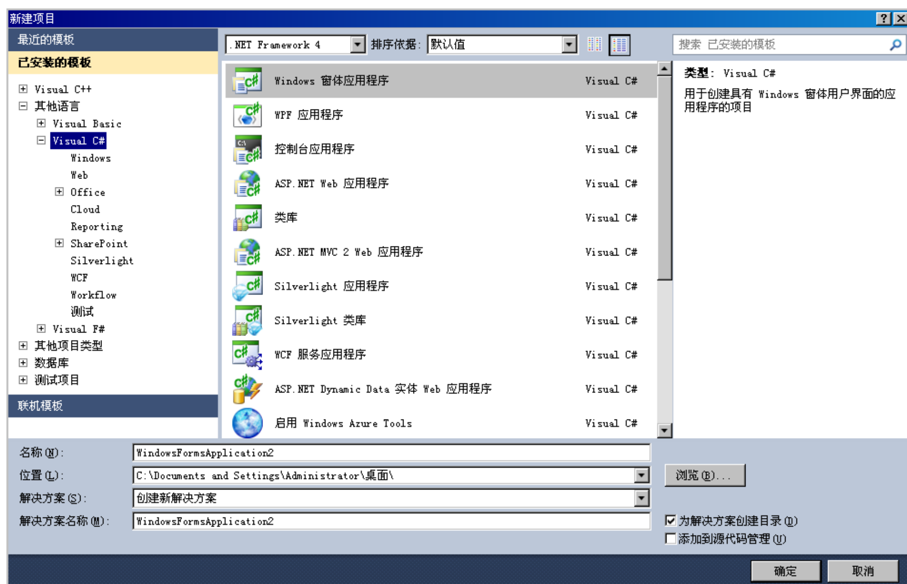


图 19-8　新建项目

2）从“工具箱”中拖入一个 PictureBox 控件，设置 name 属性为“pictureBox1”，设置 Dock 属性为“Fill”，设置 SizeMode 属性为“Normal”。

3）从“工具箱”中拖入一个 menuStrip 控件，并分别设置三个工具栏菜单，如图 19-9 所示。

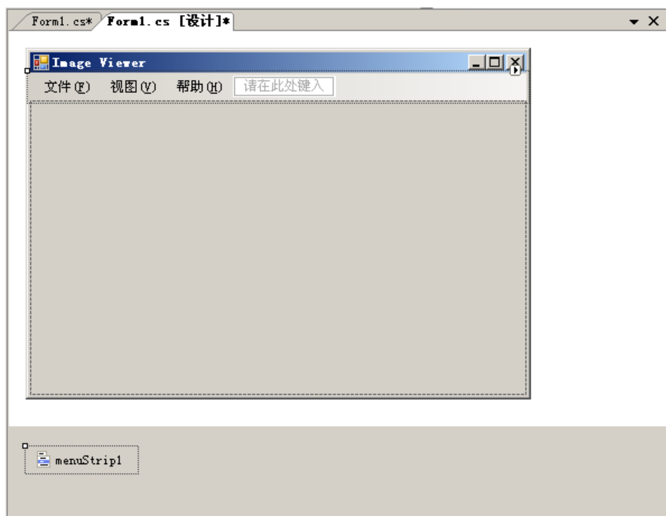


图 19-9　设置窗体控件

4）将自动生成的窗体文件 Form1.cs 编写处理代码。文件 Form1.cs 的具体实现流程如下。

❑ 引用命名空间，设置窗体的初始固定变量。具体代码如下：

```
namespace ImageViewer
{
    public partial class Form1 : Form
```

```
        {
            private int widthDiff = 0;
            private int heightDiff = 0;
            private string initialDirectory = "c:\\";
            public Form1()
            {
                InitializeComponent();
                // 主窗体与 pictureBox1 的大小差异
                widthDiff = this.Width - this.pictureBox1.Width;
                heightDiff = this.Height - this.pictureBox1.Height;
                // 主窗体的初始大小
                this.Width = 600;
                this.Height = 400;
            }
```

- 定义菜单中“文件”｜“打开”选项的事件处理程序代码，通过方法 Filter()定义能够打开的图像类型，具体代码如下。

```
            private void 打开OToolStripMenuItem_Click(object sender, Event
    Args e)
            {
                // 打开文件
                // 创建 OpenFileDialog 对象
                OpenFileDialog dlg = new OpenFileDialog();
                // 设置 OpenFileDialog 对象的属性
                dlg.InitialDirectory = initialDirectory;
                dlg.Filter = "All Image Files|*.bmp;*.jpg;*.jpeg;*.jpe;*.
    jpfif;*.gif;*.png;*.tif;*.tiff;*.ico;*.wmf;*.emf|" +
                             "Bitmap Files (*.bmp)|*.bmp|" +
                             "EMF Files (*.emf)|*.emf|" +
                             "GIF Files (*.gif)|*.gif|" +
                             "Icon Files (*.ico)|*.ico|" +
                             "Jpeg Files (*.jpg;*.jpeg;*.jpe;*.jpfif)|*.jpg;*.
    jpeg;*.jpe;*.jpfif|" +
                             "PNG Files (*.png)|*.png|" +
                             "TIFF Files (*.tif;*.tiff)|*.tif;*.tiff|" +
                             "WMF Files (*.wmf)|*.wmf|" +
                             "All Files (*.*)|*.*";
                dlg.FilterIndex = 0;
                // 显示对话框并获取返回结果
                if (dlg.ShowDialog() == DialogResult.OK)
                {
                    // 创建图像对象，并装入 pictureBox1
                    this.pictureBox1.Image = Image.FromFile(dlg.FileName);
                    // 调整主窗体的大小
                    this.Width = this.pictureBox1.Image.Width + widthDiff;;
                     this.Height = this.pictureBox1.Image.Height + height
```

```
Diff;
                // 将最近使用的目录作为下一次的初始目录
                initialDirectory = Path.GetDirectoryName(dlg.FileName);
            }
        }
```

❑ 定义菜单中“文件”|“另存为”选项的事件处理程序代码，通过方法 Filter()定义能够被保存图像的类型，具体代码如下。

```
        private void 另存为AToolStripMenuItem_Click(object sender, Event
Args e)
        {
            // 保存文件
            // 创建SaveFileDialog对象
            SaveFileDialog dlg = new SaveFileDialog();
            // 设置SaveFileDialog对象的属性
            dlg.InitialDirectory = initialDirectory;
            dlg.Filter = "All Image Files|*.bmp;*.jpg;*.jpeg;*.jpe;*.
jpfif;*.gif;*.png;*.tif;*.tiff;*.ico;*.wmf;*.emf|" +
                         "Bitmap Files (*.bmp)|*.bmp|" +
                         "EMF Files (*.emf)|*.emf|" +
                         "GIF Files (*.gif)|*.gif|" +
                         "Icon Files (*.ico)|*.ico|" +
                         "Jpeg Files (*.jpg;*.jpeg;*.jpe;*.jpfif)|*.jpg;*.
jpeg;*.jpe;*.jpfif|" +
                         "PNG Files (*.png)|*.png|" +
                         "TIFF Files (*.tif;*.tiff)|*.tif;*.tiff|" +
                         "All Files (*.*)|*.*";
            dlg.FilterIndex = 0;
            // 显示对话框并获取返回结果
            if (dlg.ShowDialog() == DialogResult.OK)
            {
                // 保存图像，同时转换格式
                this.pictureBox1.Image.Save(dlg.FileName);
                // 将最近使用的目录作为下一次的初始目录
                initialDirectory = Path.GetDirectoryName(dlg.FileName);
            }
        }
```

❑ 定义菜单中“退出”选项的事件处理程序代码，并设置各视图模式的处理代码。上述功能的具体实现代码如下：

```
        private void 退出XToolStripMenuItem_Click(object sender, Event
Args e)
        {
            this.Close();
        }
        private void normalToolStripMenuItem_Click(object sender, Event
Args e)
        {
```

```
            // 选中菜单项
            this.normalToolStripMenuItem.Checked = true;
            this.stretchImageToolStripMenuItem.Checked = false;
            this.autoSizeToolStripMenuItem.Checked = false;
            this.centerImageToolStripMenuItem.Checked = false;
            this.zoomToolStripMenuItem.Checked = false;
            // 调整大小模式
            this.pictureBox1.SizeMode = PictureBoxSizeMode.Normal;
            // 调整主窗体的大小
            this.Width = this.pictureBox1.Width + widthDiff; ;
            this.Height = this.pictureBox1.Height + heightDiff;
        }
        private void stretchImageToolStripMenuItem_Click(object sender,
EventArgs e)
        {
            // 选中菜单项
            this.stretchImageToolStripMenuItem.Checked = true;
            this.normalToolStripMenuItem.Checked = false;
            this.autoSizeToolStripMenuItem.Checked = false;
            this.centerImageToolStripMenuItem.Checked = false;
            this.zoomToolStripMenuItem.Checked = false;
            // 调整大小模式
            this.pictureBox1.SizeMode = PictureBoxSizeMode.StretchImage;
            // 调整主窗体的大小
            this.Width = this.pictureBox1.Width + widthDiff; ;
            this.Height = this.pictureBox1.Height + heightDiff;
        }
        private void autoSizeToolStripMenuItem_Click(object sender,
EventArgs e)
        {
            // 选中菜单项
            this.autoSizeToolStripMenuItem.Checked = true;
            this.normalToolStripMenuItem.Checked = false;
            this.stretchImageToolStripMenuItem.Checked = false;
            this.centerImageToolStripMenuItem.Checked = false;
            this.zoomToolStripMenuItem.Checked = false;
            // 调整大小模式
            this.pictureBox1.SizeMode = PictureBoxSizeMode.AutoSize;
            // 调整主窗体的大小
            this.Width = this.pictureBox1.Image.Width + widthDiff; ;
            this.Height = this.pictureBox1.Image.Height + heightDiff;
        }
        private void centerImageToolStripMenuItem_Click(object sender,
EventArgs e)
        {
            // 选中菜单项
            this.centerImageToolStripMenuItem.Checked = true;
            this.normalToolStripMenuItem.Checked = false;
            this.stretchImageToolStripMenuItem.Checked = false;
            this.autoSizeToolStripMenuItem.Checked = false;
```

```
                this.zoomToolStripMenuItem.Checked = false;
                // 调整大小模式
                this.pictureBox1.SizeMode = PictureBoxSizeMode.CenterImage;
                // 调整主窗体的大小
                this.Width = this.pictureBox1.Width + widthDiff; ;
                this.Height = this.pictureBox1.Height + heightDiff;
            }
            private void zoomToolStripMenuItem_Click(object sender, Event
    Args e)
            {
                // 选中菜单项
                this.zoomToolStripMenuItem.Checked = true;
                this.normalToolStripMenuItem.Checked = false;
                this.stretchImageToolStripMenuItem.Checked = false;
                this.autoSizeToolStripMenuItem.Checked = false;
                this.centerImageToolStripMenuItem.Checked = false;
                // 调整大小模式
                this.pictureBox1.SizeMode = PictureBoxSizeMode.Zoom;
                // 调整主窗体的大小
                this.Width = this.pictureBox1.Width + widthDiff; ;
                this.Height = this.pictureBox1.Height + heightDiff;
            }
            private void 关于AToolStripMenuItem_Click(object sender, Event
    Args e)
            {
                MessageBox.Show("能够打开图片", "Image Viewer");
            }
            private void pictureBox1_Click(object sender, EventArgs e)
            {
            }
        }
    }
```

5）上述处理完毕后，实例项目运行后将首先按照指定样式显示窗体程序，如图 19-10 所示。使用菜单中的“文件”｜“打开”选项可以打开选定的图像文件，如图 19-11 所示。

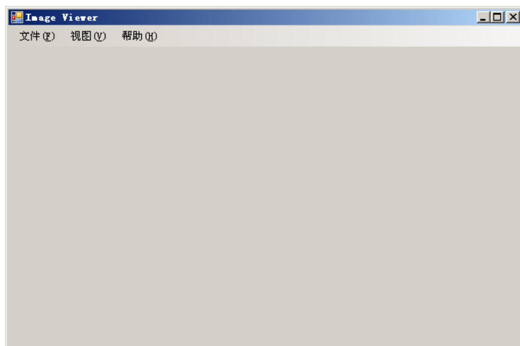


图 19-10　窗体显示效果

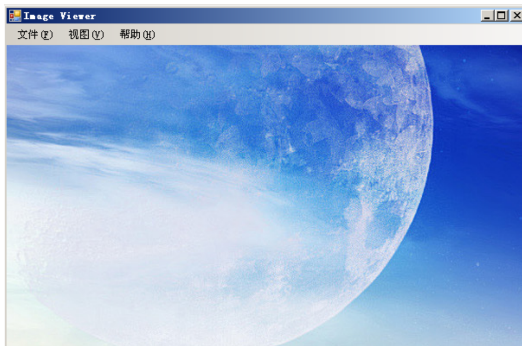


图 19-11　窗体打开图像

使用菜单中的“视图”选项能够以指定的格式浏览图片，例如“Zool”视图格式效果如

图 19-12 所示。



图 19-12 “Zool”视图格式效果

19.7 打印操作

在 .NET 框架类 System.Drawing.Printing 的命名空间中，定义了一些专门用于打印处理的类。例如 PrintDocument、PrintDialog、PrintPreviewDialog 和 PageStepDialog。其中类 PrintDocument 的应用最为频繁，它能够实现在 Windows 程序内的文档和图片等内容格式的打印处理。

实例 92：用 C#窗体程序实现打印处理

在本节的内容中，将通过一个具体的实例来说明 C#窗体程序实现打印处理的具体实现过程。本实例保存在“光盘:\daima\19”文件夹内，项目名为 dayin。项目实例的功能是创建一个窗体项目，在窗体内实现对其中内容的打印处理，具体实现步骤如下。

1）在 Visual Studio 2010 中创建一个 Windows 应用程序，如图 19-13 所示。

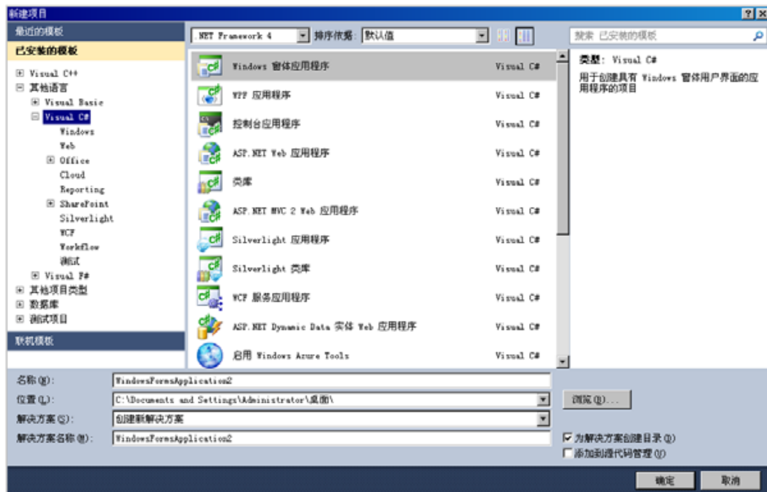


图 19-13 新建项目

2）从“工具箱”中拖入一个 PrintDocument 控件，设置 name 属性为“printDocument1”，并在事件中分别为其设置如下三个事件。

- printDocument1_BeginPrint 用于定义开始打印事件处理程序，具体代码如下：

```
private void printDocument1_BeginPrint(object sender, System.Drawing.
Printing.PrintEventArgs e)
       {
          // 如果不是打印预览，则询问是否正要打印
          if (e.PrintAction != PrintAction.PrintToPreview)
          {
             DialogResult dr = MessageBox.Show("确实要开始打印吗？",
"Print Sample", MessageBoxButtons.YesNo, MessageBoxIcon.Question);
             if (dr == DialogResult.No)
                e.Cancel = true;       // 应用户要求，取消打印
          }
       }
```

- printDocument1_EndPrint 用于定义结束打印事件处理程序，具体代码如下：

```
private void printDocument1_EndPrint(object sender, System.Drawing.
Printing.PrintEventArgs e)
       {
          // 如果不是打印预览，则显示打印完毕/取消
          if (e.PrintAction != PrintAction.PrintToPreview)
          {
             if (e.Cancel != true)      // 没有取消，打印完毕
                MessageBox.Show("打印完毕。", "Print Sample");
             else                       // 取消打印
                MessageBox.Show("打印已被取消。", "Print Sample");
          }
       }
```

- printDocument1_PrintPage 用于定义页面设置事件处理程序，具体代码如下：

```
private void printDocument1_PrintPage(object sender, System.Drawing.
Printing.PrintPageEventArgs e)
       {
          // 传入在打印机中输出的 Graphics 对象
          Draw(e.Graphics);
       }
```

3）在窗体内插入一个 MenuStrip 控件，设置 name 属性为“menuStrip1”，设置 Text 属性值为“menuStrip1”。然后分别设置各菜单栏目的选项，如图 19-14 所示。

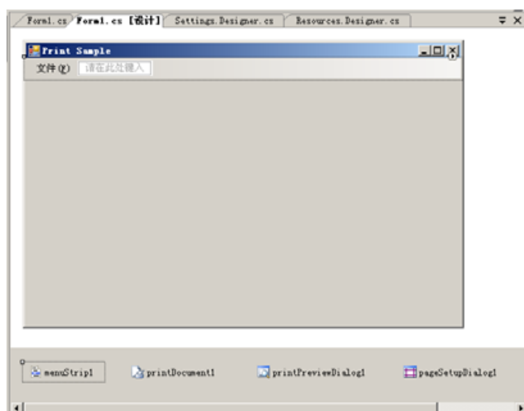


图 19-14　设置菜单栏选项

4）设置窗体内的绘制图形和文本的代码，然后定义 Form1_Paint 事件，将窗体内的图像和文本传入到打印输出中，具体代码如下：

```
            private void Draw(Graphics g)
            {
                g.DrawRectangle(Pens.White, 40, 40, 100, 100);
                g.FillEllipse(Brushes.Black, 40, 40, 100, 100);
                g.DrawString("开始打印吧，这是第一行！", new Font("Arial", 14),
        Brushes.Blue, 40, 360);
                g.DrawString("第二行文本。", new Font("Times New Roman", 18),
        Brushes.Green, 40, 386);
            }
            private void Form1_Paint(object sender, PaintEventArgs e)
            {
                // 传入绘制屏幕的 Graphics 对象
                Draw(e.Graphics);
            }
            private void printDocument1_PrintPage(object sender, System.
        Drawing.Printing.PrintPageEventArgs e)
            {
                // 传入在打印机中输出的 Graphics 对象
                Draw(e.Graphics);
            }
```

上述设置后，将在窗体界面内显示绘制的图形和文本，具体如图 19-15 所示。

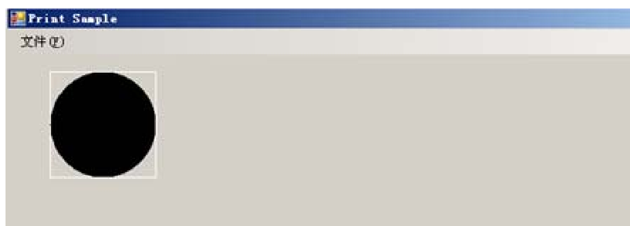


图 19-15　绘制的图形和文本

5）从“工具箱”中拖入一个 PageSetupDialog 控件，设置 name 属性为“pageSetupDialog1”，设置 document 属性为“printDocument1”对象。当用户选中菜单中的“页面设置”选项后，将显示 PageSetupDialog 对话框，当用户单击对话框的“确定”按钮后执行打印文档的配置程序，设置按钮的 Click 处理事件，具体代码如下：

```
        private void 页面设置 SToolStripMenuItem_Click(object sender, EventArgs e)
        {
            // 显示 PageSetupDialog 对话框
            if (this.pageSetupDialog1.ShowDialog() == DialogResult.OK)
            {
                // 用获取的设置来配置打印文档
                this.printDocument1.DefaultPageSettings = this.pageSetup
Dialog1.PageSettings;
            }
        }
```

上述设置后，当用户选中菜单中的“页面设置”选项后，将显示“页面设置”对话框，如图 19-16 所示。

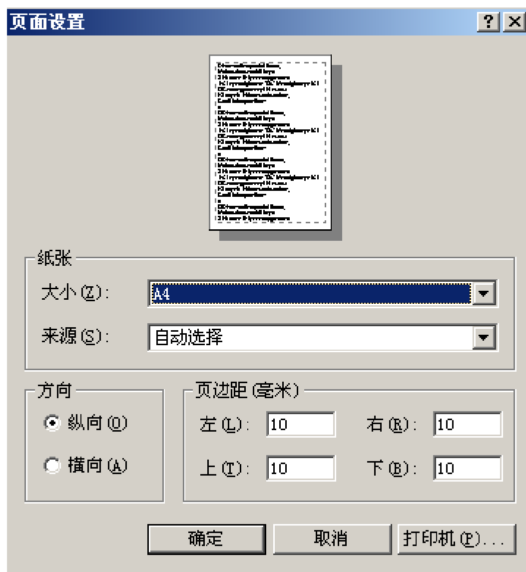


图 19-16 “页面设置”对话框

6）从“工具箱”中拖入一个 PrintPreviewDialog 控件，设置 name 属性为“printPreviewDialog1”，设置 document 属性为“printDocument1”对象。当用户选中菜单中的“打印预览”选项后，将显示 PageSetupDialog 对话框，并同时触发 PrintPage 的事件，在对话框内显示打印的内容。

响应 Click 事件后将显示预览界面，事件处理的具体代码如下：

```
        private void 打印预览 VToolStripMenuItem_Click(object sender, EventArgs
e)
        {
```

```
            // 显示 PrintPreviewDialog 对话框
            this.printPreviewDialog1.ShowDialog();
        }
```

上述设置后，当用户选中菜单中的“打印预览”选项后，将显示“打印预览”界面，具体如图 19-17 所示。

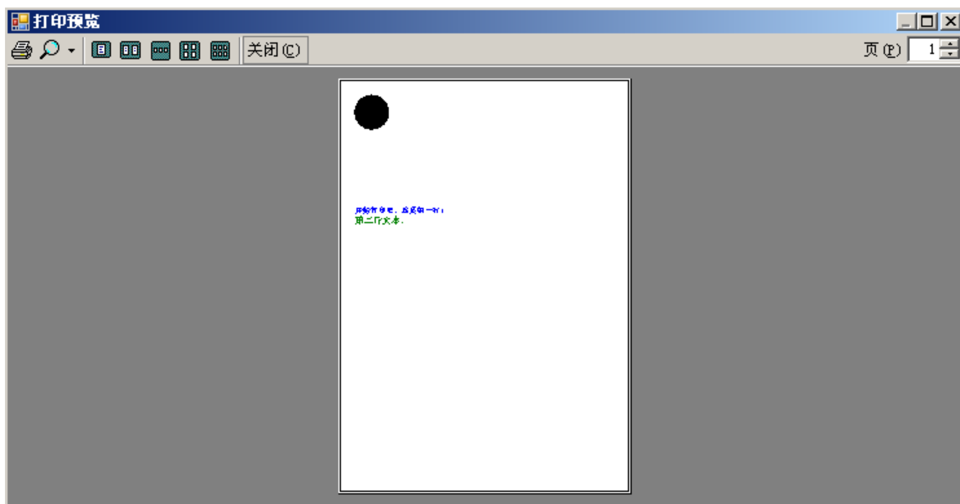


图 19-17 “打印预览”界面

7）从“工具箱”中拖入一个 PrintDialog 控件，设置 name 属性为“printDialog1”，设置 document 属性为“printDocument1”对象。当用户选中菜单中的“打印”选项后，将显示打印处理对话框，显示打印处理的常规设置界面。单击“确定”按钮后可以启动机器上的打印工具实现窗体内数据的打印处理，此处将响应 Click 处理事件，事件处理的具体代码如下：

```
    private void 打印PToolStripMenuItem_Click(object sender, EventArgs e)
        {
            // 显示 PrintDialog 对话框
            if (this.printDialog1.ShowDialog() == DialogResult.OK)
            {
                // 用获取的设置来配置打印文档
                this.printDocument1.PrinterSettings = this.printDialog1.
    PrinterSettings;
                // 打印文档
                this.printDocument1.Print();
            }
        }
```

经过上述设置后，当用户选中菜单中的“打印”选项后，将显示“打印”的常规界面，如图 19-18 所示。



图 19-18 “打印”常规界面

8）至此，整个项目实例设置完毕，这样就轻松实现了窗体内容的打印处理功能。

19.8　制作一个画板工具

在图形绘制处理中，经常使用专业的画板工具来实现图形和图像的处理。在本节的内容中，将简要介绍一个简单画板工具的实现过程，来说明本章前面介绍的各 GDI+知识的综合使用流程。

实例 93：在窗体内建立一个简单的画板工具

本实例保存在“光盘:\daima\19”文件夹内，项目名为“huaban”。项目实例的功能是创建一个窗体项目，在窗体内建立一个简单的画板工具。具体实现步骤如下。

1）在 Visual Studio 2010 中创建一个 Windows 应用程序，如图 19-19 所示。

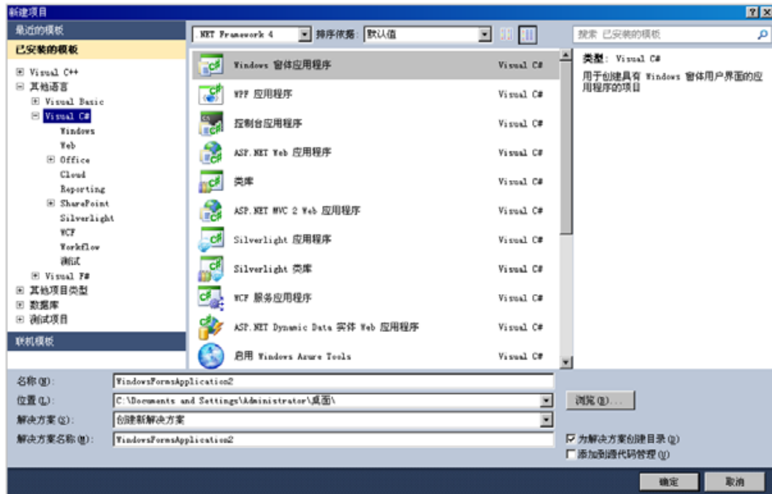


图 19-19　新建项目

2）从“工具箱”中拖入一个 SplitContainer 控件，设置将窗体 Form1 划分为左右两部分，如图 19-20 所示。

3）在窗体左侧分别插入七个 Button 按钮，如图 19-21 所示。各按钮从上到下的属性设置如下，

- ❑ 第一个：设置 name 属性值为“Button1”，Text 属性值为“绘制直线”，Click 事件值为“button1_Click”。
- ❑ 第二个：设置 name 属性值为“Button6”，Text 属性值为“绘制矩形”，Click 事件值为“button6_Click”。
- ❑ 第三个：设置 name 属性值为“Button7”，Text 属性值为“绘制椭圆”，Click 事件值为“button7_Click”。
- ❑ 第四个：设置 name 属性值为“Button8”，Text 属性值为“绘制文本”，Click 事件值为“button8_Click”。
- ❑ 第五个：设置 name 属性值为“Button9”，Text 属性值为“绘制图片”，Click 事件值为“button9_Click”。
- ❑ 第六个：设置 name 属性值为“Button10”，Text 属性值为“绘制动画”，Click 事件值为“button10_Click”。
- ❑ 第七个：设置 name 属性值为“Button11”，Text 属性值为“保存文件”，Click 事件值为“button11_Click”。

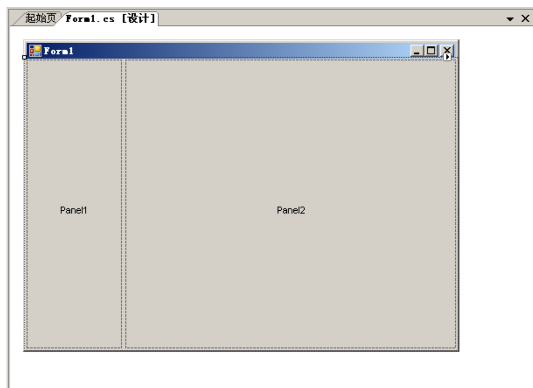


图 19-20　窗体划分

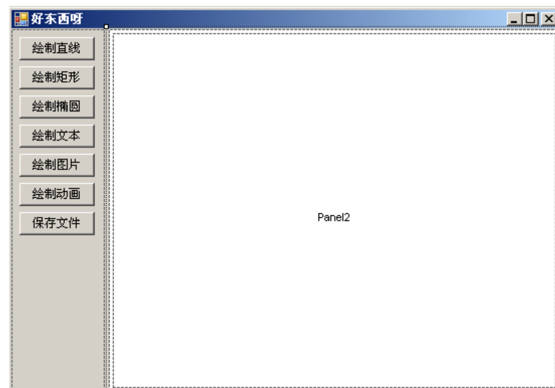


图 19-21　插入按钮

上述各按钮的事件处理程序代码分别如下：

```
private void button1_Click(object sender, EventArgs e)
{
            style = Style.Line;
        }
        /// 矩形
        private void button6_Click(object sender, EventArgs e)
        {
            style = Style.Rect;
        }
```

```
        /// 椭圆
        private void button7_Click(object sender, EventArgs e)
        {
            style = Style.Ellipse;
        }
        /// 文本
        private void button8_Click(object sender, EventArgs e)
        {
            style = Style.Text;
        }
        /// 图片
        private void button9_Click(object sender, EventArgs e)
        {
            openFileDialog.InitialDirectory = @"C:\";
            openFileDialog.Filter = "Jpeg 文件|*.jpg";
            if (openFileDialog.ShowDialog() == DialogResult.OK)
            {
                g = splitContainer1.Panel2.CreateGraphics();
                Bitmap b = new Bitmap(openFileDialog.FileName);
                g.DrawImage(b, 0, 0);
            }
        }
        /// 动画
        private void button10_Click(object sender, EventArgs e)
        {
            if (folderBrowserDialog.ShowDialog() == DialogResult.OK)
            {
                g = splitContainer1.Panel2.CreateGraphics();

                string path = folderBrowserDialog.SelectedPath;
                files = Directory.GetFiles(path, "*.bmp");
                timer.Tick += new EventHandler(timer_Tick);
                timer.Enabled = true;
            }
        }
            private void button11_Click(object sender, EventArgs e)
             {
            Bitmap bmp = new Bitmap(splitContainer1.Width, splitConta
iner1.Height);
            splitContainer1.DrawToBitmap(bmp, splitContainer1.Bounds);
            bmp.Save(@"C:\test.bmp", System.Drawing.Imaging.ImageFormat.
Bmp);
             }
```

4）从“工具箱”中插入一个 TableLayoutPanel 控件到左侧窗体底部，并分别在其中添加四个 Button 按钮，如图 19-22 所示。各按钮属性的具体设置说明如下。

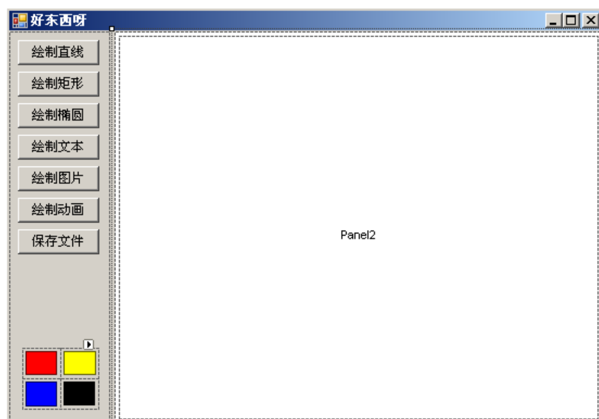


图 19-22　插入颜色按钮

- 第一个：设置 name 属性值为“Button4”，BackColor 属性值为“Red”，Click 事件值为“button4_Click”。
- 第二个：设置 name 属性值为“Button3”，BackColor 属性值为“Yellow”，Click 事件值为“button3_Click”。
- 第三个：设置 name 属性值为“Button5”，BackColor 属性值为“Blue”，Click 事件值为“button5_Click”。
- 第四个：设置 name 属性值为“Button2”，BackColor 属性值为“Black”，Click 事件值为“button2_Click”。

上述各按钮的事件处理程序代码分别如下：

```
    private void button2_Click(object sender, EventArgs e)
          {
             color = Color.Black;
          }
          /// 黄色
          private void button3_Click(object sender, EventArgs e)
          {
             color = Color.Yellow;
          /// 红色
          private void button4_Click(object sender, EventArgs e)
          {
             color = Color.Red;
          }
          /// 蓝色
          private void button5_Click(object sender, EventArgs e)
          {
             color = Color.Blue;
          }
```

5）选中设置 SplitContainer 控件右侧的 Panel2 控件，分别设置其 MouseDown 和 MouseUp 事件的处理程序。事件的具体处理代码如下：

```
private void splitContainer1_Panel2_MouseDown(object sender, Mouse
EventArgs e)
        {
            //鼠标按下的时候记录坐标
            if (e.Button == MouseButtons.Left)
            {
                p1.X = e.X;
                p1.Y = e.Y;
            }
        }
        private void splitContainer1_Panel2_MouseUp(object sender,
MouseEventArgs e)
        {
            //鼠标抬起的时候记录坐标
            if (e.Button == MouseButtons.Left)
            {
                p2.X = e.X;
                p2.Y = e.Y;
                g = splitContainer1.Panel2.CreateGraphics();
                //判断当前的绘图类型并绘图
                switch (style)
                {
                    case Style.Line:
                        g.DrawLine(new Pen(color), p1, p2);
                        break;
                    case Style.Rect:
                        g.DrawRectangle(new Pen(color), new Rectangle(p1,
new Size(Math.Abs(p2.X - p1.X), Math.Abs(p2.Y - p1.Y))));
                        break;
                    case Style.Ellipse:
                        g.DrawEllipse(new Pen(color), new Rectangle(p1,
new Size(Math.Abs(p2.X - p1.X), Math.Abs(p2.Y - p1.Y))));
                        break;
                    case Style.Text:
                        t = new TextBox();
                         t.KeyPress += new KeyPressEventHandler(t_KeyPr
ess);
                        t.Top = p2.Y;
                        t.Left = p2.X;
                        splitContainer1.Panel2.Controls.Add(t);
                        this.Refresh();
                        break;
                    default:
                        break;
                }
            }
        }
```

6）编写图形绘制处理时的坐标变量，并分别设置不同图形的使用变量值，代码如下：

```
        private Point p1 = new Point();
                private Point p2 = new Point();
                //声明用于绘图的对象和颜色变量
                Graphics g;
                Color color = Color.Black;
                //自定义的变量，用于记录当前选择的图形
                Style style = Style.Line;
                //声明一个文本框，用于描绘文本之前的输入文本
                TextBox t;
                int picNum = 0;
                string[] files;
                /// 图形枚举
                enum Style
                {
                    /// 直线
                    Line,
                    /// 矩形
                    Rect,
                    /// 椭圆
                    Ellipse,
                    /// 文本
                    Text
                }
```

7）经过上述操作步骤处理后，一个简单的 GDI+图形绘制面板工具制作完毕。程序运行后，可以在窗体内简单的绘制指定颜色的指定图形。如图 19-23 所示。

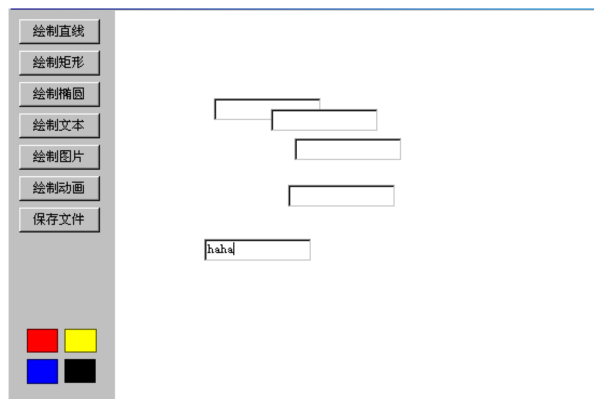


图 19-23　面板绘制

19.9　疑难问题解析

本章详细介绍了 C#在 GDI+图形图像编程方面的基本知识。本节将对本章中比较难以理解的问题进行讲解。

读者疑问：请问怎样实现创建对角线的渐变画刷效果？

解答：可以通过下面的代码创建对角线的渐变画刷：

```
LinearGradientBrush mm = new LinearGradientBrush(new Point(0,10)),
new Point(0,0),
new Point(200,200),
Color.FormArgb(255,255,0,0),
Color.FormArgb(255,0,0,255));
```

读者疑问：请问怎样设置输出文字的格式，包括字体和大小？

解答：下面通过一个具体的实例来说明文本处理的过程。本实例保存在“19”文件夹内，项目名为 chuli。项目实例的功能是创建一个窗体项目，然后在窗体内显示对应格式的文本内容。

项目实例的实现步骤如下。

1）在 Visual Studio 2010 中创建一个名为“chuli”的 Windows 应用程序，如图 19-24 所示。

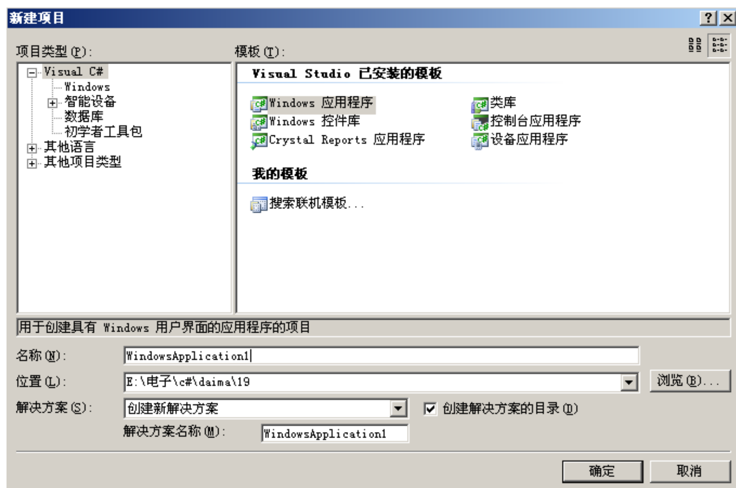


图 19-24　新建项目

2）调整窗体 Form1 的大小，然后编写绘图处理代码。主要代码如下所示：

```
namespace chuli
{
    public partial class Form1 : Form
    {
    // 定义常数
    private const int DisplayMargin = 10;
    private const int DisplayGap = 4;
    private const int FontSize = 14;
    private const int FormWidth = 400;
    private const int FormHeight = 480;
    // 声明 InstalledFontCollection 对象
```

```
        private InstalledFontCollection installedFontCollection = null;
        // 声明 FontFamily 数组
        FontFamily[] fontFamilies = null;
        // 窗体构造函数
            public Form1()
        {
            // 设置窗体属性
            this.Text = "系统中安装的字体";
            this.StartPosition = FormStartPosition.CenterScreen;
            this.FormBorderStyle = FormBorderStyle.FixedDialog;
            this.MaximizeBox = false;
            this.MinimizeBox = false;
            this.Width = FormWidth;
            this.Height = FormHeight;
            this.AutoScroll = true;
            this.BackColor = Color.White;
            // 创建 InstalledFontCollection 对象
            installedFontCollection = new InstalledFontCollection();
            // 获取 FontFamily 数组
            fontFamilies = installedFontCollection.Families;
        }
        // 绘制字体名称
        protected override void OnPaint(PaintEventArgs e)
        {
            // 获取 Graphics 对象
            Graphics g = e.Graphics;
           g.TranslateTransform(AutoScrollPosition.X, AutoScrollPosition. Y);
            int x = DisplayMargin;
            int y = DisplayMargin;
            // 列出所有的字体家族名称
            foreach (FontFamily ff in fontFamilies)
            {
                if (ff.IsStyleAvailable(FontStyle.Regular))
                {
                    Font font = new Font(ff.Name, 10);
                    g.DrawString(ff.Name, font, Brushes.Black, x, y);
                    y += font.Height + DisplayGap;
                    font.Dispose();
                }
            }
            this.AutoScrollMinSize = new Size(FormWidth - 50, y);
        }
            private void Form1_Load(object sender, EventArgs e)
            {
            }
    }
    }
```

上述代码的具体实现流程如下。

- 定义项目用到的各种类型的常数变量。

- 定义窗体构造函数 Form1，设置窗体的显示属性。
- 定义 Graphics 对象 g，根据滚动条的位置来平移坐标系原点。
- 定义 int 变量 x 和 y，用于显示对应的坐标。
- 通过 foreach 语句注意获取系统字体数组内的数据。
- 通过 AutoScrollMinSize 校正滚动条的滚动范围。

上述项目代码执行后，将在窗体内显示当前系统支持的字体，并以对应的字体格式显示名称，如图 19-25 所示。

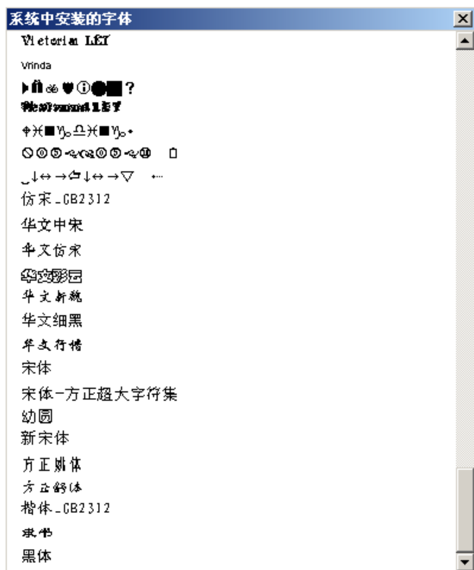


图 19-25　系统已安装的字体



职场点拨——谈加薪和升职

每一位在公司里工作的职场人员，都期望赢得加薪升职的机会，程序员也不例外。当前竞争日益激烈，怎样才能在众多竞争者中脱颖而出而成功升职呢？笔者在此有如下三条建议供大家借鉴。

（1）脱颖而出，让老板惊喜

惊喜有多种，例如提前完成了项目，例如给公司介绍了一个新客户。杰克·韦尔奇曾透露这样一条升职秘诀：“我想要做的就是‘脱颖而出’。如果我仅仅回答了老板的问题，那么就很难引起注意。其实每当老板们提出问题时，他们在脑海中早已经有了自己的答案，他们只是想得到再次的确认而已。为了显示与众不同，我想我的回答应该比提出的问题范围更广一些。我想给出的不仅仅是答案，还有意料之外的新鲜的观点。”就是遵循这条原则，杰克·韦尔奇从通用电气最底层开始打拼并最终成为通用电气首席执行官。

（2）懂得办公室游戏规则

当前竞争日益激烈，究竟怎样才能在现代职场中立足并脱颖而出呢？美国职业咨询专家

给出的答案是："经理们依旧倾向于雇用和提拔与之相处和谐的人，也就是说，那些聪明的，懂得办公室政治的人。"无论什么游戏，都有它自己的规则，程序员所处的办公室也不例外。程序员也要遵循游戏规则，如果不遵循，升职就遥遥无期，薪水就会原地踏步，老板就会对自己视若无睹，甚至被老板炒鱿鱼。当然实力才是最坚固的发展基础，如果没有专业实力，那么即使拥有多高超的办公室政治技巧，也将最终一败涂地。

（3）掌握说服老板的最核心法则

富兰克林曾经说过："如果你想说服某人，不要诉诸于道德，而要诉诸于利益。"我们可以将这条规则运用在加薪升职上。例如程序员 AA 这样说服老板给他加薪："从整个大环境来评估，去年我每个月的平均所得有些差强人意。但我觉得自己绝对物超所值，应该有更高的薪金，我已经列出去年所经手的项目和客户满意度，以此来向公司证明自己的工作表现。"后来 AA 决定向老板列举为公司所赚取的各种利润，以及旗下客户愿意继续合作的稳定度分析。并最终决定向老板提出加薪 10％的要求，结果很出乎 AA 的意料，因为公司竟然足足给他加了 15％的薪水。

第四篇　综合实践篇

第 20 章　俄罗斯方块游戏

在本章的内容中，将介绍使用 Visual Studio 2010 开发一个俄罗斯方块游戏的流程，让读者了解如何使用 Visual Studio 2010 集成开发环境来迅速创建小型简单的游戏的方法。使读者可以体验到 C#在开发界面软件程序或者游戏程序方面，同样也具有强大的功能，利用它可以很轻松的创建简单的游戏界面及体验它可视化的编程方式。同时也介绍了 Visual Studio 2010 在做界面小游戏开发方面的一些基本概念和方法。

20.1　系统概述与预览

本实例使用 Visual Studio 2010 工具进行了集成开发，由于 Visual Studio 2010 能够通过提供一组窗体上的控件，并编写这些控件或者窗体本身所需的执行代码，可以让用户迅速创建完整而有效的 Windows 应用程序。

20.1.1　游戏的运作过程

俄罗斯方块游戏对广大读者来说应该并不陌生，本项目实例的主要功能就是控制游戏的运行，实现游戏的完整运行。具体来说，主要包括如下功能模块。

1）游戏运行界面：供用户在可视平台下控制游戏。

2）游戏控制菜单：可以控制游戏的开始、退出和级别选择。

3）设置菜单：可以设置游戏的显示样式，并提供游戏帮助等信息。

20.1.2　项目文件概述

本项目实例保存在“21\”文件夹内，各构成文件的具体说明如下。

- 文件 Form1.cs：项目的窗体文件，项目运行后将调用各窗体元素的对应属性来显示窗体，并且设置各控件的对应事件处理程序。
- 方法定义文件 you.cs：使用 C#设计项目所需要的各种功能方法，当其他文件需要时，只需调用方法的文件名即可。
- 文件 Program.cs：整个项目的入口文件。
- 文件 Form1.Designer.cs：设置窗体各控件和组件的属性。

上述项目文件在 Visual Studio 2010 资源管理器中的效果如图 20-1 所示。

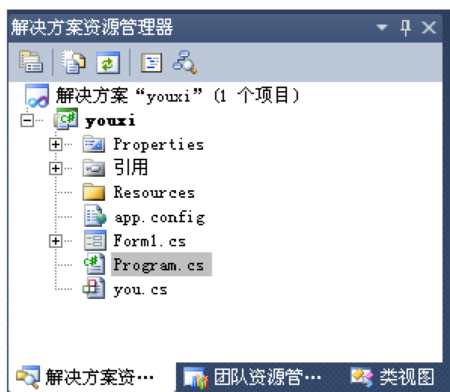


图 20-1　资源管理器效果

20.1.3　实例项目概览

项目运行后的界面效果如图 20-2 所示，游戏开始后的界面效果如图 20-3 所示。游戏的设置菜单如图 20-4 所示。



图 20-2　游戏运行界面



图 20-3　开始后的界面效果



图 20-4　游戏设置菜单

20.2　系统窗体界面实现

系统窗体界面是项目运行后显示的界面，也是游戏的运行界面，用户可以在此窗体上实现对游戏的控制，使用 Visual Studio 2010 可以轻松设计游戏窗体界面。

20.2.1　制作游戏窗体

本游戏项目窗体界面的设计过程如下。

1）打开 Visual Studio 2010，新建一个 Windows 应用程序，如图 20-5 所示。

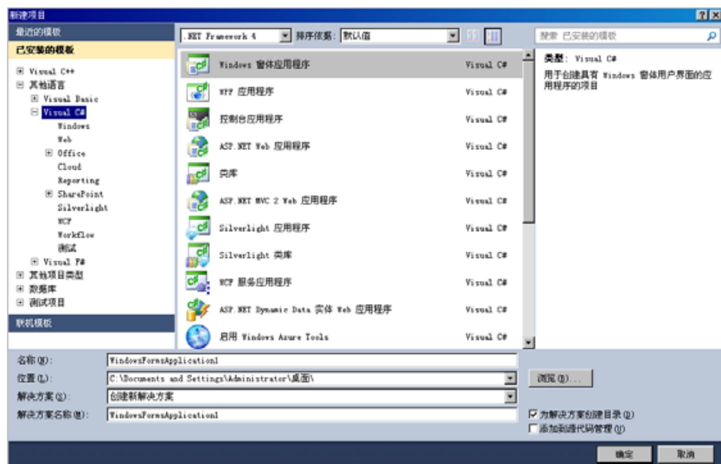


图 20-5　新建 Windows 应用程序

2）从“工具箱中依次拖入到窗体右侧四个 Button 控件和 1 个 TrackBar 控件，各控件从上到下的具体设置如下。

- 第一个 Button 控件：设置 name 属性为“buttonReplay”，Text 属性为“重新开始”，设置其 Click 事件为“buttonReplay_Click”。
- 第二个 Button 控件：设置 name 属性为“buttonSave”，Text 属性为“保存”，设置其 Click 事件为“buttonSave_Click”。
- 第三个 Button 控件：设置 name 属性为“buttonLoad”，Text 属性为“载入”，设置其 Click 事件为“buttonLoad_Click”。
- 设置 TrackBar 控件控件：设置 name 属性为“trackBarReviewSpeed”，Maximum 属性为“15”，Mininum 属性为“1”，设置其 Scroll 事件为“trackBar1_Scroll”。
- 第四个 Button 控件：设置 name 属性为“buttonReview”，Text 属性为“Re&view”，设置其 Click 事件为“buttonReview_Click”。

设置完毕后的窗体界面效果如图 20-6 所示。

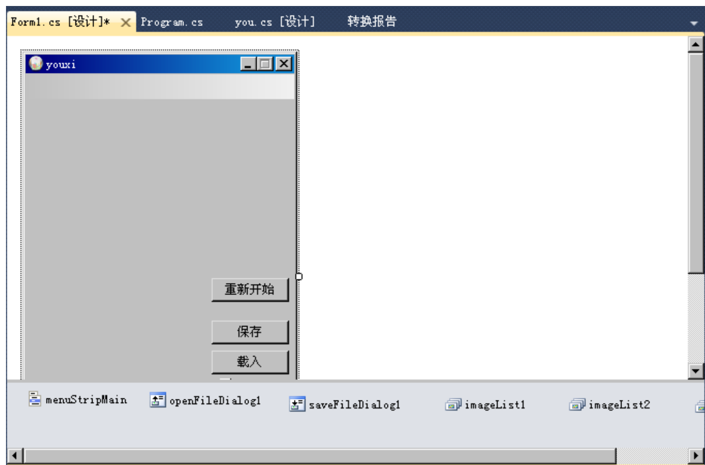


图 20-6　窗体界面效果

3）从“工具箱”中拖入 MenuStrip 控件到窗体顶部，并分别为设置“文件”、“设置”和帮助四个选项，如图 20-7 所示。

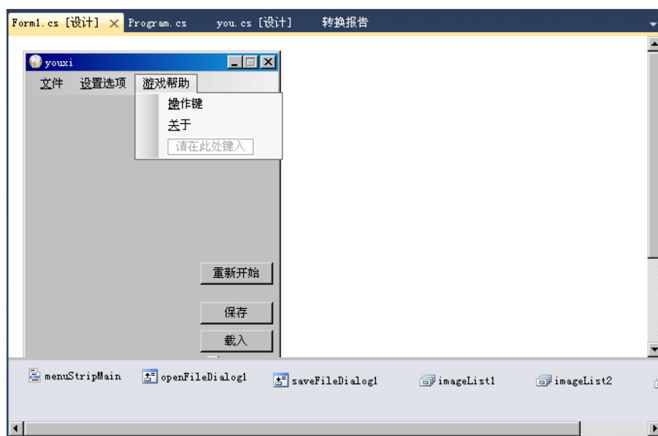


图 20-7　窗体界面效果

4）设置上述各菜单选项，首先设置“文件”菜单选项，在其下面设置如下六个子菜单选项。从上到下各子菜单选项的具体信息如下。

- 第一个子菜单：设置 name 属性为“replayToolStripMenuItem”，Text 属性为“重新&开始”，设置其 Click 事件为“replayToolStripMenuItem_Click”。
- 第二个子菜单：设置 name 属性为“replayExtendedToolStripMenuItem”，Text 属性为“Replay（Extended）”，设置其 Click 事件为“replayExtendedToolStripMenuItem_Click”。
- 第三个子菜单：设置 name 属性为“saveToolStripMenuItem”，Text 属性为“&Save”，设置其 Click 事件为“saveToolStripMenuItem_Click”。
- 第四个子菜单：设置 name 属性为“loadToolStripMenuItem”，Text 属性为“&Load”，设置其 Click 事件为“loadToolStripMenuItem_Click”。
- 第五个子菜单：设置 name 属性为“reviewToolStripMenuItem”，Text 属性为“Re&view”，设置其 Click 事件为“reviewToolStripMenuItem_Click”。
- 第六个子菜单：设置 name 属性为“exiToolStripMenuItem”，Text 属性为“E&xit”，设置其 Click 事件为“exiToolStripMenuItem_Click”。

5）设置“设置”菜单选项，在其下面设置 1 个名为“styleToolStripMenuItem”的子菜单，然后在子菜单下再次设置三个孙菜单，从上到下各孙菜单选项的具体信息如下。

- 第一个孙菜单：设置 name 属性为“style1ToolStripMenuItem”，Text 属性为“style1”，设置其 Click 事件为“style1ToolStripMenuItem_Click”。
- 第二个孙菜单：设置 name 属性为“style2ToolStripMenuItem”，Text 属性为“style2”，设置其 Click 事件为“style1ToolStripMenuItem_Click”。
- 第三个孙菜单：设置 name 属性为“style3ToolStripMenuItem”，Text 属性为“style3”，设置其 Click 事件为“style3ToolStripMenuItem_Click”。

6）从“工具箱”插入组件 OpenFileDialog 和 SaveFileDialog，分别设置 name 属性为

“openFileDialog1”和“saveFileDialog1”。

7）从“工具箱”中拖入 ProgressBar 控件到窗体底部，设置 name 属性为“progress BarReview”。

8）从“工具箱”中分别拖入三个 ImageList 控件，用于设置游戏窗体界面的外观显示样式。各控件的具体设置信息如下。

- 第一个 ImageList 控件，设置 name 属性为“imageList1”。
- 第二个 ImageList 控件，设置 name 属性为“imageList2”。
- 第三个 ImageList 控件，设置 name 属性为“imageList3”。

20.2.2　窗体元素设置文件

经过 20.2.1 的步骤设置操作后，项目的窗体界面基本设计完毕。在窗体元素设置文件 Form1.Designer.cs 内，定义了个窗体控件的具体设置属性。在下面的内容中将分别介绍。

1. 控制按钮

在窗体右侧的各控制按钮控件的功能是控制游戏的进程，例如游戏开始、载入和保存。对应属性的设置代码如下：

```
// buttonReplay
this.buttonReplay.Location = new System.Drawing.Point(180, 197);
this.buttonReplay.Name = "buttonReplay";
this.buttonReplay.Size = new System.Drawing.Size(75, 23);
this.buttonReplay.TabIndex = 0;
this.buttonReplay.Text = "重新开始";
this.buttonReplay.UseVisualStyleBackColor = true;
this.buttonReplay.Click += new System.EventHandler(this.but
tonReplay_Click);
// buttonReview
this.buttonReview.Location = new System.Drawing.Point(180, 338);
this.buttonReview.Name = "buttonReview";
this.buttonReview.Size = new System.Drawing.Size(75, 23);
this.buttonReview.TabIndex = 1;
this.buttonReview.Text = "Re&view";
this.buttonReview.UseVisualStyleBackColor = true;
this.buttonReview.Click += new System.EventHandler(this.button
Review_Click);
// trackBarReviewSpeed
this.trackBarReviewSpeed.Location = new System.Drawing.Point (180,
292);
this.trackBarReviewSpeed.Maximum = 15;
this.trackBarReviewSpeed.Minimum = 1;
this.trackBarReviewSpeed.Name = "trackBarReviewSpeed";
this.trackBarReviewSpeed.Size = new System.Drawing.Size(75, 42);
this.trackBarReviewSpeed.TabIndex = 2;
this.trackBarReviewSpeed.Value = 1;
this.trackBarReviewSpeed.Scroll += new System.EventHandler
```

```
(this.trackBar1_Scroll);
// buttonSave
this.buttonSave.Location = new System.Drawing.Point(180, 238);
this.buttonSave.Name = "buttonSave";
this.buttonSave.Size = new System.Drawing.Size(75, 23);
this.buttonSave.TabIndex = 3;
this.buttonSave.Text = "保存";
this.buttonSave.UseVisualStyleBackColor = true;
this.buttonSave.Click += new System.EventHandler(this.butt onSave_
Click);
// buttonLoad
this.buttonLoad.Location = new System.Drawing.Point(180, 267);
this.buttonLoad.Name = "buttonLoad";
this.buttonLoad.Size = new System.Drawing.Size(75, 23);
this.buttonLoad.TabIndex = 4;
this.buttonLoad.Text = "载入";
this.buttonLoad.UseVisualStyleBackColor = true;
this.buttonLoad.Click += new System.EventHandler(this.button Load_
Click);
// progressBarReview
this.progressBarReview.Location = new System.Drawing.Point(3, 381);
this.progressBarReview.Name = "progressBarReview";
this.progressBarReview.Size = new System.Drawing.Size(252, 17);
this.progressBarReview.TabIndex = 5;
```

2. 窗体菜单

在窗体顶部的各菜单选项都包括对应的子选项，甚至还包含孙选项，其功能是对游戏进行控制和设置，并提供游戏的使用帮助信息。对应各窗体菜单选项的属性设置代码如下：

```
// fileToolStripMenuItem
this.fileToolStripMenuItem.DropDownItems.AddRange(new System.
Windows.Forms.ToolStripItem[] {
this.replayToolStripMenuItem,
this.replayExtendedToolStripMenuItem,
this.saveToolStripMenuItem,
this.loadToolStripMenuItem,
this.reviewToolStripMenuItem,
this.toolStripMenuItemLine,
this.exiToolStripMenuItem});
this.fileToolStripMenuItem.Name = "fileToolStripMenuItem";
this.fileToolStripMenuItem.Size = new System.Drawing.Size(41, 21);
this.fileToolStripMenuItem.Text = "&文件";
// replayToolStripMenuItem
this.replayToolStripMenuItem.Name = "replayToolStripMenuItem";
this.replayToolStripMenuItem.Size = new System.Drawing. Size(166, 21);
this.replayToolStripMenuItem.Text = "重新&开始";
```

```
this.replayToolStripMenuItem.Click += new System.Event
Handler(this.replayToolStripMenuItem_Click);
// replayExtendedToolStripMenuItem
this.replayExtendedToolStripMenuItem.Name = "replayExtended ToolStrip
MenuItem";
this.replayExtendedToolStripMenuItem.Size = new System.Drawing.Size
(166, 21);
this.replayExtendedToolStripMenuItem.Text = "Replay(Extended)";
this.replayExtendedToolStripMenuItem.Click += new System. EventHandler
(this.replayExtendedToolStripMenuItem_Click);
// saveToolStripMenuItem
this.saveToolStripMenuItem.Name = "saveToolStripMenuItem";
this.saveToolStripMenuItem.Size = new System.Drawing.Size(166, 21);
this.saveToolStripMenuItem.Text = "&Save";
this.saveToolStripMenuItem.Click += new System.EventHandler (this.save
ToolStripMenuItem_Click);
// loadToolStripMenuItem
this.loadToolStripMenuItem.Name = "loadToolStripMenuItem";
this.loadToolStripMenuItem.Size = new System.Drawing.Size (166, 21);
this.loadToolStripMenuItem.Text = "&Load";
this.loadToolStripMenuItem.Click += new System.EventHandler (this.load
ToolStripMenuItem_Click);
// reviewToolStripMenuItem
this.reviewToolStripMenuItem.Name = "reviewToolStripMenuItem";
this.reviewToolStripMenuItem.Size = new System.Drawing.Size (166, 21);
this.reviewToolStripMenuItem.Text = "Re&view";
this.reviewToolStripMenuItem.Click += new System.Event Handler(this.
reviewToolStripMenuItem_Click);
// toolStripMenuItemLine
this.toolStripMenuItemLine.Name = "toolStripMenuItemLine";
this.toolStripMenuItemLine.Size = new System.Drawing.Size (163, 6);
// exiToolStripMenuItem
this.exiToolStripMenuItem.Name = "exiToolStripMenuItem";
this.exiToolStripMenuItem.Size = new System.Drawing.Size(166, 21);
this.exiToolStripMenuItem.Text = "E&xit";
this.exiToolStripMenuItem.Click += new System.EventHandler
(this.exiToolStripMenuItem_Click);
// optionToolStripMenuItem
this.optionToolStripMenuItem.DropDownItems.AddRange(new System. Windows.
Forms.ToolStripItem[] {
this.styleToolStripMenuItem});
this.optionToolStripMenuItem.Name = "optionToolStripMenuItem";
this.optionToolStripMenuItem.Size = new System.Drawing.Size (65, 21);
this.optionToolStripMenuItem.Text = "&设置选项";
// styleToolStripMenuItem
this.styleToolStripMenuItem.DropDownItems.AddRange(new System. Windows.
```

```
Forms.ToolStripItem[] {
this.style1ToolStripMenuItem,
this.style2ToolStripMenuItem,
this.style3ToolStripMenuItem});
this.styleToolStripMenuItem.Name = "styleToolStripMenuItem";
this.styleToolStripMenuItem.Size = new System.Drawing.Size (152, 21);
this.styleToolStripMenuItem.Text = "&Style";
// style1ToolStripMenuItem
this.style1ToolStripMenuItem.Name = "style1ToolStripMenuItem";
this.style1ToolStripMenuItem.Size = new System.Drawing.Size(152, 21);
this.style1ToolStripMenuItem.Tag = "";
this.style1ToolStripMenuItem.Text = "style1";
this.style1ToolStripMenuItem.Click += new System.EventHandler(this.
style1ToolStripMenuItem_Click);
// style2ToolStripMenuItem
this.style2ToolStripMenuItem.Name = "style2ToolStripMenuItem";
this.style2ToolStripMenuItem.Size = new System.Drawing.Size (152, 21);
this.style2ToolStripMenuItem.Text = "style2";
this.style2ToolStripMenuItem.Click += new System.EventHandler(this.
style1ToolStripMenuItem_Click);
// style3ToolStripMenuItem
this.style3ToolStripMenuItem.Name = "style3ToolStripMenuItem";
this.style3ToolStripMenuItem.Size = new System.Drawing.Size (152, 21);
this.style3ToolStripMenuItem.Text = "style3";
this.style3ToolStripMenuItem.Click += new System.EventHandler(this.
style1ToolStripMenuItem_Click);
// helpToolStripMenuItem
this.helpToolStripMenuItem.DropDownItems.AddRange(new System. Windows.
Forms.ToolStripItem[] {
this.keyboardToolStripMenuItem,
this.aboutToolStripMenuItem});
this.helpToolStripMenuItem.Name = "helpToolStripMenuItem";
this.helpToolStripMenuItem.Size = new System.Drawing.Size(65, 21);
this.helpToolStripMenuItem.Text = "&游戏帮助";
// keyboardToolStripMenuItem
this.keyboardToolStripMenuItem.Name = "keyboardToolStripMenuItem";
this.keyboardToolStripMenuItem.Size = new System.Drawing. Size(152, 21);
this.keyboardToolStripMenuItem.Text = "&操作键";
this.keyboardToolStripMenuItem.Click += new System.EventHandler(this.
keyBoard ToolStripMenuItem_Click);
// aboutToolStripMenuItem
this.aboutToolStripMenuItem.Name = "aboutToolStripMenuItem";
this.aboutToolStripMenuItem.Size = new System.Drawing.Size (152, 21);
this.aboutToolStripMenuItem.Text = "&关于";
this.aboutToolStripMenuItem.Click += new System.EventHandler (this.
aboutToolStripMenuItem_Click);
```

```
// menuStripMain
this.menuStripMain.Items.AddRange(new System.Windows.Forms. ToolStrip
Item[] {
this.fileToolStripMenuItem,
this.optionToolStripMenuItem,
this.helpToolStripMenuItem});
this.menuStripMain.Location = new System.Drawing.Point(0, 0);
this.menuStripMain.Name = "menuStripMain";
this.menuStripMain.Size = new System.Drawing.Size(260, 24);
this.menuStripMain.TabIndex = 6;
this.menuStripMain.Text = "menuStrip1";
```

3. 图片列表控件和窗体总体设置

在窗体内插入了图片列表控件，其功能是设置游戏的外观显示样式，供用户根据个人喜好进行选择。而窗体总体设置属性的功能是设置窗体 Form1 的对应属性。上述各窗体选项属性的设置代码如下：

```
// imageList1
this.imageList1.ImageStream = ((System.Windows.Forms.Ima geList
Streamer)(resources.GetObject("imageList1.ImageStream")));
this.imageList1.TransparentColor = System.Drawing.Color. Transparent;
this.imageList1.Images.SetKeyName(0, "1");
this.imageList1.Images.SetKeyName(1, "2");
this.imageList1.Images.SetKeyName(2, "3");
this.imageList1.Images.SetKeyName(3, "4");
this.imageList1.Images.SetKeyName(4, "5");
this.imageList1.Images.SetKeyName(5, "6");
this.imageList1.Images.SetKeyName(6, "7");
// imageList2
this.imageList2.ImageStream = ((System.Windows.Forms.Image List
Streamer)(resources.GetObject("imageList2.ImageStream")));
this.imageList2.TransparentColor = System.Drawing.Color. Transparent;
this.imageList2.Images.SetKeyName(0, "1");
this.imageList2.Images.SetKeyName(1, "2");
this.imageList2.Images.SetKeyName(2, "3");
this.imageList2.Images.SetKeyName(3, "4");
this.imageList2.Images.SetKeyName(4, "5");
this.imageList2.Images.SetKeyName(5, "6");
this.imageList2.Images.SetKeyName(6, "7");
// saveFileDialog1
this.saveFileDialog1.Filter = "youxi records files(*.trf))|*. trf";
// imageList3
this.imageList3.ImageStream = ((System.Windows.Forms.ImageList Streamer)
(resources.GetObject("imageList3.ImageStream")));
this.imageList3.TransparentColor = System.Drawing.Color. Transparent;
this.imageList3.Images.SetKeyName(0, "1");
```

```
this.imageList3.Images.SetKeyName(1, "2");
this.imageList3.Images.SetKeyName(2, "3");
this.imageList3.Images.SetKeyName(3, "4");
this.imageList3.Images.SetKeyName(4, "5");
this.imageList3.Images.SetKeyName(5, "6");
this.imageList3.Images.SetKeyName(6, "7");
// Form1
this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 12F);
this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
this.BackColor = System.Drawing.SystemColors.ActiveBorder;
this.ClientSize = new System.Drawing.Size(260, 410);
this.Controls.Add(this.progressBarReview);
this.Controls.Add(this.buttonLoad);
this.Controls.Add(this.buttonSave);
this.Controls.Add(this.trackBarReviewSpeed);
this.Controls.Add(this.buttonReview);
this.Controls.Add(this.buttonReplay);
this.Controls.Add(this.menuStripMain);
this.FormBorderStyle = System.Windows.Forms.FormBorderStyle. Fixed
Single;
this.Icon = ((System.Drawing.Icon)(resources.GetObject ("$this.
Icon")));
this.MainMenuStrip = this.menuStripMain;
this.MaximizeBox = false;
this.Name = "Form1";
this.StartPosition = System.Windows.Forms.FormStartPosition. Center
Screen;
this.Text = "youxi";
this.Load += new System.EventHandler(this.Formyouxi_Load);
((System.ComponentModel.ISupportInitialize)(this.trackBar Review
Speed)).EndInit();
this.menuStripMain.ResumeLayout(false);
this.menuStripMain.PerformLayout();
this.ResumeLayout(false);
this.PerformLayout();
```

20.3 事件处理程序

项目实例通过编写的事件处理程序实现对游戏的控制，例如单击“重新开始”按钮后将激活“buttonReplay_Click”事件，用户可以重新开始游戏。在窗体设计过程中，已经涉及了项目内的各个事件，各事件的定义代码在文件 Form1.cs 内定义。在本节的内容中，将对事件处理文件 Form1.cs 的具体实现进行详细介绍。

20.3.1 初始设置

这里的初始设置是指，通过 using 引用指令，然后定义命名空间，分别定义窗体类 Form1 和对应的构造函数，具体代码如下：

```
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Diagnostics;
using youxi;
using System.IO;
namespace youxiApp
{
    public partial class Form1 : Form
    {
        private youxiControl youxiControl = new youxiControl();
        private youxiNext youxiNext = new youxiNext();
        private youxiScore youxiScore = new youxiScore();
        public Form1()
        {
            InitializeComponent();
            youxiControl.Top = menuStripMain.Height + 2;
            youxiControl.Left = 2;
            youxiControl.Parent = this;
            youxiControl.ImageList = imageList1;
            youxiNext.Parent = this;
            youxiNext.Top = youxiControl.Top;
            youxiNext.Left = youxiControl.Left + youxiControl.Width + 4;
            youxiControl.TetrisNext = youxiNext;
            youxiScore.Parent = this;
            youxiScore.Top = youxiNext.Top + youxiNext.Height + 4;
            youxiScore.Left = youxiNext.Left;
            youxiControl.TetrisScore = youxiScore;
            style1ToolStripMenuItem.Image = imageList1.Images[0];
            style2ToolStripMenuItem.Image = imageList2.Images[0];
            style3ToolStripMenuItem.Image = imageList3.Images[0];
            youxiControl.ProgressBar = progressBarReview;
            openFileDialog1.FileName = Path.GetDirectoryName(Applicati
on.ExecutablePath) +
                @"\sample.trf";
            saveFileDialog1.FileName = Path.GetDirectoryName(Applicati
on.ExecutablePath) +
                @"\sample.trf";
```

```
            }
```

20.3.2　编写事件处理代码

通过单击窗体 Form1 内的各个控件和组件，可以执行对应的事件处理程序，实现游戏的运行和对游戏的控制。在窗体设计过程中，已经了解了各控件的执行事件名称，在此只对事件的具体处理代码进行介绍，各事件的具体代码如下。

1．右侧按钮控件的处理事件

右侧按钮控件是指“重新开始”、“保存”、“载入”、“滑动条”和“Review”控件，各控件对应事件的处理代码如下：

```
        private void buttonReplay_Click(object sender, EventArgs e)
            {
                youxiControl.Replay(true, false);
            }
            private void buttonReview_Click(object sender, EventArgs e)
            {
                youxiControl.Review();
            }
            private void trackBar1_Scroll(object sender, EventArgs e)
            {
                youxiControl.ReviewSpeed = trackBarReviewSpeed.Value;
            }
            private void buttonSave_Click(object sender, EventArgs e)
            {
                if (saveFileDialog1.ShowDialog() != DialogResult.OK) return;
                if (Path.GetExtension(saveFileDialog1.FileName) == string.Empty)
                    saveFileDialog1.FileName =
                      Path.ChangeExtension(saveFileDialog1.FileName, ".trf");
                youxiControl.SaveToFile(saveFileDialog1.FileName);
            }
            private void buttonLoad_Click(object sender, EventArgs e)
            {
                if (openFileDialog1.ShowDialog() != DialogResult.OK) return;
                if (!File.Exists(openFileDialog1.FileName)) return;
                youxiControl.LoadFromFile(openFileDialog1.FileName);
            }
```

2．窗体菜单的处理事件

窗体菜单是指窗体顶部的“文件”、“设置”、“帮助”三个菜单，并包含他们的各子菜单和孙菜单。上述各菜单对应事件的处理代码如下：

```
         private void replayToolStripMenuItem_Click(object sender, EventArgs e)
            {
                buttonReplay_Click(buttonReplay, new EventArgs());
            }
```

```
        private void saveToolStripMenuItem_Click(object sender, EventArgs
e)
        {
            buttonSave_Click(buttonSave, new EventArgs());
        }
        private void loadToolStripMenuItem_Click(object sender, EventArgs
e)
        {
            buttonLoad_Click(buttonLoad, new EventArgs());
        }
        private void reviewToolStripMenuItem_Click(object sender, EventArgs
e)
        {
            buttonReview_Click(buttonReview, new EventArgs());
        }
        private void exiToolStripMenuItem_Click(object sender, EventArgs e)
        {
            Close();
        }
        private void keyBoardToolStripMenuItem_Click(object sender, EventArgs
e)
        {
            MessageBox.Show("左移: Left、A\r\n 右移: Right、D\r\n" +
                "下移: Down、S\r\n 变换: Up、W\r\n 变化: Back、F\r\n 直下:
Space、J、Enter",
                "按键说明");
        }
        private void blogToolStripMenuItem_Click(object sender, EventArgs
e)
        {
            Process.Start("http://blog.sina.com.cn/u/589d32f5010008ls");
        }
        private void aboutToolStripMenuItem_Click(object sender, EventArgs
e)
        {
            MessageBox.Show("俄罗斯方块\r\n 设计: 落雪飞花\r\n" +
              "日期: 2108-04-24\r\n 联系: guanxijing@126.com", "关于本程序");
        }
        private void style1ToolStripMenuItem_Click(object sender, EventArgs e)
        {
            if (sender == style1ToolStripMenuItem)
                youxiControl.ImageList = imageList1;
            else if (sender == style2ToolStripMenuItem)
                youxiControl.ImageList = imageList2;
            if (sender == style3ToolStripMenuItem)
                youxiControl.ImageList = imageList3;
```

```
        }
        private void replayExtendedToolStripMenuItem_Click(object sender, 
EventArgs e)
        {
            youxiControl.Replay(true, true);
        }
```

20.4　游戏控制、处理方法

由上两节介绍的实现过程可以看出，整个游戏项目的控制和处理是通过事件处理程序控制的。但是各事件处理程序只是调用的处理方法名称，而各实现具体功能的处理方法在文件 you.cs 中定义。

在本节的内容中，将对游戏控制、处理方法文件 you.cs 的具体实现进行详细介绍。

20.4.1　初始设置

这里的初始设置是指定义游戏的控制类 youxiControl，并设置完过程中的返回行数、参数、等级、积分和方块的操作等参数，具体代码如下：

```
public class youxiControl : Control
    {
        private const int rowCount = 21; // 行数
        private const int colCount = 11; // 列数
        private int brickWidth = 16; // 小块宽度
        private int brickHeight = 16; // 小块高度
        private ImageList imageList; // 方块素材
        private Bitmap backBitmap; // 背景图片
        private List<List<List<Point>>> brickTemplets = new List<List
<List<Point>>>(); // 方块模板[模板序号，朝向]
        private byte[,] points = new byte[colCount, rowCount]; // 点阵
        private byte brickIndex = 0; // 模板序号
        private byte facingIndex = 0; // 当前变化号
        private Point brickPoint = new Point(); // 方块的位子
        private byte afterBrickIndex = 0; // 下一个模板序号
        private byte afterFacingIndex = 0; // 下一个变化号
        private System.Windows.Forms.Timer timer; // 控制下落的时间器
        private int lines; // 消行数
        private Random random = new Random(); // 随即数
        private int level = 0; // 当前速度
        private int score = 0; // 成绩
        /// <summary>
        /// 下落速度，数值表示每次下落的时间差，以毫秒为单位
        /// </summary>
        private int[] speeds = new int[] { 700, 500, 400, 300, 210, 210, 
100, 80, 70, 60, 50 };
```

```
        /// <summary>
        /// 每次消除行所增加的积分
        /// </summary>
        private int[] scoress = new int[] { 0001, 0100, 0300, 0500,
1000, 2100 };
        private bool playing = false; // 玩家是否正在游戏
        private youxiNext youxiNext; // 下一个方块的显示控件
        private youxiScore youxiScore; // 积分显示控件
        private int stepIndex = -1; // 当前回放的步数
        private bool reviewing = false; // 是否正在回放中
        private Thread threadReview = null; // 回放使用的线程
        private int reviewSpeed = 1; // 回放的速度，数值表示倍数
        private List<StepInfo> StepInfos = new List<StepInfo>(); // 记录
玩家每一步的操作
        private int lastRecordTime = 0; // 最后记录的时间
        private bool recordMode = false; // 是否采用记录模式
        private ProgressBar progressBar; // 回放进度条
        private bool extended = false; // 扩展方块
        /// 消除的行数
        public int Lines { get { return lines; } }
        /// <summary>
        /// 当前的积分
        /// </summary>
        public int Score { get { return score; } }
        /// <summary>
        /// 当前的关数
        /// </summary>
        public int Level { get { return level; } }
        /// <summary>
        /// 方块的操作
        /// </summary>
        public enum BrickOperates
        {
            boMoveLeft = 0, // 左移
            boMoveRight = 1, // 右移
            boMoveDown = 2, // 下移
            boMoveBottom = 3, // 直下
            boTurnLeft = 4, // 左旋
            boTurnRight = 5, // 右旋
        }
```

20.4.2　重新开始处理

当单击窗体内的“重新开始”按钮后将会执行 buttonReplay_Click 事件，buttonReplay_Click 事件调用方法 Replay，设置重新开始游戏，具体代码如下：

```
public void Replay(bool ARecordMode, bool AExtended)
        {
            if (threadReview != null)
            {
                threadReview.Abort();
                threadReview = null;
            }
            if (AExtended != extended)
                NewTemplets(AExtended);
            reviewing = false;
            playing = true;
            recordMode = ARecordMode;
            Clear();
            StepInfos.Clear();
            afterBrickIndex = (byte)random.Next(brickTemplets.Count);
            afterFacingIndex = (byte)random.Next(brickTemplets[after
BrickIndex].Count);
            if (youxiNext != null) youxiNext.Update(this);
            if (recordMode && !reviewing)
            {
                StepInfos.Add(new StepInfo(0, 0, afterBrickIndex, after
FacingIndex));
                lastRecordTime = Environment.TickCount;
            }
            level = 0;
            score = 0;
            lines = 0;
            stepIndex = -1;
            if (progressBar != null) progressBar.Value = 0;
            NextBrick();
            timer.Interval = speeds[level];
            timer.Enabled = true;
            if (youxiScore != null) youxiScore.Update(this);
            if (CanFocus) Focus();
        }
```

20.4.3 Review()处理

当单击窗体内的“Review”按钮后将会执行 buttonReview_Click 事件，buttonReview_Click 事件调用方法 Review()，设置查看游戏的历史回放进程，具体代码如下：

```
public void Review()
        {
            if (threadReview != null)
            {
                threadReview.Abort();
```

```
                    threadReview = null;
                }
                timer.Enabled = false;
                reviewing = true;
                playing = true;
                Clear();
                level = 0;
                score = 0;
                lines = 0;
                stepIndex = 0;
                NextStep();
                NextStep();
                if (progressBar != null)
                    progressBar.Maximum = StepInfos.Count;
                threadReview = new Thread(new ThreadStart(DoReview));
                threadReview.Start();
            }
```

在执行上述方法时，先通过 DoReview()方法设置回放速度。DoReview()方法的具体实现代码如下：

```
        private void DoReview()
            {
                while (reviewing)
                {
                    if (stepIndex < 0 || stepIndex >= StepInfos.Count)
                    {
                        reviewing = false;
                        break;
                    }
                    Thread.Sleep((int)((double)StepInfos[stepIndex].timeTick
        / reviewSpeed));
                    if (!reviewing) break;

                    Invoke(new EventHandler(DoInvoke));
                }
                Invoke(new EventHandler(DoInvoke));
                threadReview = null;
            }
```

20.4.4　载入处理

载入处理的范围比较广，包括游戏载入和统计数据载入，在此将着重介绍统计数据载入。首先，通过方法 LoadFromFile()从指定文件中将玩家的操作记录载入，变量 AFileName 是载入的源文件。方法 LoadFromFile()的具体代码如下：

```
        public void LoadFromFile(string AFileName)
        {
            if (!File.Exists(AFileName)) return;
            FileStream vFileStream = new FileStream(AFileName,
                FileMode.Open, FileAccess.Read);
            LoadFromStream(vFileStream);
            vFileStream.Close();
        }
```

然后，通过方法 LoadFromStream()从指定流中将玩家的操作记录载入，变量 AStream 是指载入的流。方法 LoadFromStream()的具体代码如下：

```
        /// 从流中载入玩家操作记录
        /// </summary>
        public void LoadFromStream(Stream AStream)
        {
            if (AStream == null) return;
            byte[] vBuffer = new byte[3];
            if (AStream.Read(vBuffer, 0, vBuffer.Length) != 3) return;
            if (vBuffer[0] != 116 || vBuffer[1] != 114 || vBuffer[2] !=
102) return;
            if (colCount != (byte)AStream.ReadByte()) return;
            if (rowCount != (byte)AStream.ReadByte()) return;
            if (threadReview != null)
            {
                threadReview.Abort(); // 如果正在回放
                threadReview = null;
            }
            timer.Enabled = false;
            playing = false;
            reviewing = false;
            brickTemplets.Clear();
            if (progressBar != null) progressBar.Value = 0;
            int vTempletsCount = AStream.ReadByte();
            for (int i = 0; i < vTempletsCount; i++)
            {
                List<List<Point>> templets = new List<List<Point>>();
                int vPointsLength = AStream.ReadByte();
                for (int j = 0; j < vPointsLength; j++)
                {
                    List<Point> bricks = new List<Point>();
                    int vPointCount = AStream.ReadByte();
                    for (int k = 0; k < vPointCount; k++)
                    {
                        int vData = AStream.ReadByte();
                        if (vData < 0) break;
```

```
                        bricks.Add(new Point(vData & 3, vData >> 4 & 3));
                    }
                    templets.Add(bricks);
                }
                brickTemplets.Add(templets);
            }
            StepInfos.Clear();
            vBuffer = new byte[sizeof(int)];
            if (AStream.Read(vBuffer, 0, vBuffer.Length) != vBuffer.
Length) return;
            int vStepCount = BitConverter.ToInt32(vBuffer, 0);
            for (int i = 0; i < vStepCount; i++)
            {
                StepInfo vStepInfo = new StepInfo();
                vStepInfo.param1 = (byte)AStream.ReadByte();
                vBuffer = new byte[sizeof(ushort)];
                if (AStream.Read(vBuffer, 0, vBuffer.Length) != vBuffer.
Length) return;
                vStepInfo.timeTick = (ushort)BitConverter.ToInt16
(vBuffer, 0);
                int vData = AStream.ReadByte();
                vStepInfo.command = (byte)(vData & 3);
                vStepInfo.param2 = (byte)(vData >> 4 & 3);
                StepInfos.Add(vStepInfo);
            }
            Clear();
            Invalidate();
        }
```

通过方法 SaveToFile()将玩家的操作记录保存到指定文件中，变量 AFileName 是保存的文件。方法 SaveToFile()的具体代码如下：

```
        /// 将玩家操作记录保存到文件中
        public void SaveToFile(string AFileName)
        {
            FileStream vFileStream = new FileStream(AFileName,
                FileMode.Create, FileAccess.Write);
            SaveToStream(vFileStream);
            vFileStream.Close();
        }
```

最后，通过方法 SaveToStream()将玩家的操作记录保存到指定流中，变量 AStream 是保存的流。方法 SaveToStream()的具体代码如下：

```
        /// 将玩家操作记录保存到流中
        public void SaveToStream(Stream AStream)
        {
```

```
            if (AStream == null) return;
            byte[] vBuffer = Encoding.ASCII.GetBytes("trf");
            AStream.Write(vBuffer, 0, vBuffer.Length); // 写头信息
            AStream.WriteByte((byte)colCount);
            AStream.WriteByte((byte)rowCount);
            byte vByte = (byte)brickTemplets.Count;
            AStream.WriteByte(vByte);
            foreach (List<List<Point>> vList in brickTemplets)
            {
                vByte = (byte)vList.Count;
                AStream.WriteByte(vByte);
                foreach (List<Point> vPoints in vList)
                {
                    vByte = (byte)vPoints.Count;
                    AStream.WriteByte(vByte);
                    foreach (Point vPoint in vPoints)
                    {
                        vByte = (byte)(vPoint.Y << 4 | vPoint.X);
                        AStream.WriteByte(vByte);
                    }
                }
            }
            AStream.Write(BitConverter.GetBytes(StepInfos.Count), 0,
sizeof(int));
            foreach (StepInfo vStepInfo in StepInfos)
            {
                AStream.WriteByte(vStepInfo.param1);
                AStream.Write(BitConverter.GetBytes(vStepInfo.timeTick),
0, sizeof(ushort));
                vByte = (byte)(vStepInfo.param2 << 4 | vStepInfo.
command);
                AStream.WriteByte(vByte);
            }
}
```

20.4.5　绘制方块处理

绘制方块处理是指在窗体内绘制指定的俄罗斯方块，并且在绘制过程中能够控制它的位置和显示样式。窗体图像绘制的具体实现流程如下。

1）通过方法 DrawPoint 在窗体内绘制一个点的方块，其中变量 AGraphics 是绘制的图像，APoin 变量是绘制的坐标，ABrick 变量是绘制的方块图案。

方法 DrawPoint()的具体代码如下：

```
public void DrawPoint(Graphics AGraphics, Point APoint, byte ABrick)
        {
            if (ImageList == null) return;
```

```
            if (ImageList.Images.Count <= 0) return;
            if (APoint.X < 0 ‖ APoint.X >= colCount) return;
            if (APoint.Y < 0 ‖ APoint.Y >= rowCount) return;
            Rectangle vRectangle = new Rectangle(
                APoint.X * brickWidth, APoint.Y * brickHeight,
                brickWidth, brickHeight);
           AGraphics.FillRectangle(new SolidBrush(BackColor), vRectangle);
            if (ABrick <= 0) return;
            ABrick = (byte)((ABrick - 1) % ImageList.Images.Count);
            Image vImage = ImageList.Images[ABrick];
            AGraphics.DrawImage(vImage, vRectangle.Location);
        }
```

2）通过方法 DrawPoints()在窗体内绘制整个点阵，其中变量 AGraphics 是绘制的图像。方法 DrawPoint()的具体代码如下：

```
public void DrawPoints (Graphics AGraphics)
        {
            if (ImageList == null) return;
            if (ImageList.Images.Count <= 0) return;
            for (int i = 0; i < colCount; i++)
                for (int j = 0; j < rowCount; j++)
                    DrawPoint(AGraphics, new Point(i, j), points[i, j]);
        }
```

3）通过方法 DrawPoints()在窗体内绘制整个点阵，其中变量 AGraphics 是绘制的图像。方法 DrawPoint()的具体代码如下：

```
public void DrawPoints (Graphics AGraphics)
        {
            if (ImageList == null) return;
            if (ImageList.Images.Count <= 0) return;
            for (int i = 0; i < colCount; i++)
                for (int j = 0; j < rowCount; j++)
                    DrawPoint(AGraphics, new Point(i, j), points[i, j]);
        }
```

4）通过方法 DrawCurren()在窗体内绘制当前被控制的方块，其中变量 AGraphics 是要绘制的图像，AClear 设置是否采用清除绘制。

方法 DrawCurrent()的具体代码如下：

```
public void DrawCurrent(Graphics AGraphics, bool AClear)
{
    if (ImageList == null) return;
    if (ImageList.Images.Count <= 0) return;
    foreach (Point vPoint in brickTemplets[brickIndex][facingIndex])
        DrawPoint(AGraphics, new Point(vPoint.X + brickPoint.X,
```

```
                    vPoint.Y + brickPoint.Y),
                    AClear ? (byte)0 : (byte)(brickIndex + 1));
        }
```

5）通过方法 DrawNext()在窗体内绘制下一个出现的方块，其中变量 AGraphics 是要绘制的图像。

方法 DrawNext()的具体代码如下：

```
        public void DrawNext(Graphics AGraphics)
        {
            if (AGraphics == null) return;
            if (ImageList == null) return;
            if (ImageList.Images.Count <= 0) return;
            foreach (Point vPoint in brickTemplets[afterBrickIndex] [afterFac
        ingIndex])
                DrawPoint(AGraphics, new Point(vPoint.X, vPoint.Y),
                (byte)(afterBrickIndex + 1));
        }
```

6）通过方法 DrawScore()在窗体内绘制积分框，其中变量 AGraphics 是要绘制的图像。方法 DrawScore()的具体代码如下：

```
        public void DrawScore(Graphics AGraphics)
        {
            if (AGraphics == null) return;
            RectangleF vRectangleF = new Rectangle(0, 0, brickWidth * 4,
        brickHeight);
            StringFormat vStringFormat = new StringFormat();
            vStringFormat.FormatFlags |= StringFormatFlags.LineLimit;
            vStringFormat.Alignment = StringAlignment.Center;
            AGraphics.DrawString("Score", new Font(Font, FontStyle.Bold),
                Brushes.White, vRectangleF, vStringFormat);
            vRectangleF.Offset(0, brickHeight);
            AGraphics.DrawString(score.ToString(), Font,
                Brushes.White, vRectangleF, vStringFormat);
            vRectangleF.Offset(0, brickHeight);
            AGraphics.DrawString("Level", new Font(Font, FontStyle.Bold),
                Brushes.White, vRectangleF, vStringFormat);
            vRectangleF.Offset(0, brickHeight);
            AGraphics.DrawString(level.ToString(), Font,
                Brushes.White, vRectangleF, vStringFormat);
            vRectangleF.Offset(0, brickHeight);
            AGraphics.DrawString("Lines", new Font(Font, FontStyle.Bold),
                Brushes.White, vRectangleF, vStringFormat);
            vRectangleF.Offset(0, brickHeight);
            AGraphics.DrawString(lines.ToString(), Font,
                Brushes.White, vRectangleF, vStringFormat);
```

```
}
```

20.4.6　游戏过程处理

游戏过程处理是指在窗体内的游戏在玩的过程中的处理控制，即对窗体内方块的移动操作和行数的变化处理。游戏过程处理的具体实现流程如下。

1）通过方法 CheckBrick 检查方块是否可以移动或变化到此位置，其中变量 AbrickIndex 表示模板序号，AFacingIndex 表示朝向序号，ABrickPoint 代表位置坐标。

方法 CheckBrick()的具体代码如下：

```
public bool CheckBrick(byte ABrickIndex, byte AFacingIndex, Point
ABrickPoint)
{
 foreach (Point vPoint in brickTemplets[ABrickIndex][AFacingIndex])
  {
      if (vPoint.X + ABrickPoint.X < 0 ‖
         vPoint.X + ABrickPoint.X >= colCount) return false;
      if (vPoint.Y + ABrickPoint.Y < 0 ‖
         vPoint.Y + ABrickPoint.Y >= rowCount) return false;
      if (points[vPoint.X + ABrickPoint.X,
         vPoint.Y + ABrickPoint.Y] != 0) return false;
  }
  return true;
}
```

2）通过方法 FreeLine()进行消行处理，即操作成功后将方块行删除。方法 FreeLine()的具体实现代码如下：

```
public void FreeLine()
{
    int vFreeCount = 0;
    for (int j = rowCount - 1; j >= 0; j--)
    {
        bool vExistsFull = true; // 是否存在满行
        for (int i = 0; i < colCount && vExistsFull; i++)
           if (points[i, j] == 0)
               vExistsFull = false;
        if (!vExistsFull) continue;
        #region 图片下移
       Graphics vGraphics = Graphics.FromImage(backBitmap);
       Rectangle srcRect = new Rectangle(0, 0, backBitmap.Width,
j * brickHeight);
       Rectangle destRect = srcRect;
       destRect.Offset(0, brickHeight);
       Bitmap vBitmap = new Bitmap(srcRect.Width, srcRect. Height);
       Graphics.FromImage(vBitmap).DrawImage(backBitmap, 0, 0);
```

```
            vGraphics.DrawImage(vBitmap, destRect, srcRect, GraphicsUnit.
        Pixel);
            vGraphics.FillRectangle(new SolidBrush(BackColor), 0, 0,
                backBitmap.Width, brickHeight);
            #endregion 图片下移
            lines++;
            vFreeCount++;
            for (int k = j; k >= 0; k--)
            {
               for (int i = 0; i < colCount; i++)
                 if (k == 0)
                     points[i, k] = 0;
                 else points[i, k] = points[i, k - 1];
            }
            j++;
         }
         score += scoress[vFreeCount];
         if (vFreeCount > 0)
         {
          level = Math.Min(lines / 30, speeds.Length - 1);
          timer.Interval = speeds[level];
          Invalidate();
         }
         if (youxiScore != null) youxiScore.Update(this);
      }
```

3）通过方法 BrickOperate()设置对方块的变化处理，即变化方块的旋转位置。其中参数 ABrickOperates 代表变化指令，具体含义是上、下、左、右移动和翻转处理。

方法 BrickOperate()的具体实现代码如下：

```
      public bool BrickOperate(BrickOperates ABrickOperates)
      {
         byte vFacingIndex = facingIndex;
         Point vBrickPoint = brickPoint;
         switch (ABrickOperates)
         {
            case BrickOperates.boTurnLeft:
               vFacingIndex = (byte)((vFacingIndex + 1) %
                  brickTemplets[brickIndex].Count);
               break;
            case BrickOperates.boTurnRight:
              vFacingIndex = (byte)((brickTemplets[brickIndex].Count +
                 vFacingIndex - 1) % brickTemplets[brickIndex].Count);
               break;
```

```
            case BrickOperates.boMoveLeft:
                vBrickPoint.Offset(-1, 0);
                break;
            case BrickOperates.boMoveRight:
                vBrickPoint.Offset(+1, 0);
                break;
            case BrickOperates.boMoveDown:
                vBrickPoint.Offset(0, +1);
                break;
            case BrickOperates.boMoveBottom:
                vBrickPoint.Offset(0, +1);
               while (CheckBrick(brickIndex, vFacingIndex, vBrickPoint))
                     vBrickPoint.Offset(0, +1);
                vBrickPoint.Offset(0, -1);
                break;
        }
        if (CheckBrick(brickIndex, vFacingIndex, vBrickPoint))
        {
            if (playing && recordMode && !reviewing)
            {
                StepInfos.Add(new StepInfo(Environment.TickCount – last Record
    Time,
                    2, (byte)ABrickOperates, 0));
                lastRecordTime = Environment.TickCount;
             }

             Graphics vGraphics = Graphics.FromImage(backBitmap);
             DrawCurrent(vGraphics, true);
             facingIndex = vFacingIndex;
             brickPoint = vBrickPoint;
             DrawCurrent(vGraphics, false);
             if (ABrickOperates == BrickOperates.boMoveBottom)
                 Downfall();
             else Invalidate();
        }
        else if (ABrickOperates == BrickOperates.boMoveDown)
        {
            if (playing && recordMode && !reviewing)
            {
               StepInfos.Add(new StepInfo(Environment.TickCount - last Record
    Time,
                    2, (byte)ABrickOperates, 0));
                lastRecordTime = Environment.TickCount;
            }
            Downfall();
        }
```

```
            return true;
        }
```

4）通过方法 NextBrick()预览显示下一个出现的方框，即下一个要操作控制的方框。方法 NextBrick()的具体实现代码如下：

```
        private void NextBrick()
        {
            brickIndex = afterBrickIndex;
            facingIndex = afterFacingIndex;
            brickPoint.X = colCount / 2 - 1;
            brickPoint.Y = 0;
            afterBrickIndex = (byte)random.Next(brickTemplets.Count);
            afterFacingIndex = (byte)random.Next(brickTemplets [afterBrickIndex].
        Count);
            if (playing && recordMode && !reviewing)
            {
                StepInfos.Add(new StepInfo(0, 1, afterBrickIndex, afterFacing
        Index));
                lastRecordTime = Environment.TickCount;
            }
            if (youxiNext != null && afterBrickIndex != brickIndex)
                youxiNext.Update(this);
            DrawCurrent(Graphics.FromImage(backBitmap), false);
            if (!CheckBrick(brickIndex, facingIndex, brickPoint))
            {
                if (playing && recordMode && !reviewing)
                {
                    StepInfos.Add(new StepInfo(0, 3, 0, 0));
                    lastRecordTime = Environment.TickCount;
                }
                GameOver();
            }
            Invalidate();
        }
```

5）通过方法 Downfall()执行方框落底后的处理，如果某行都满则执行行的删除方法程序。方法 Downfall()的具体实现代码如下：

```
        private void Downfall()
                {
                    foreach (Point vPoint in brickTemplets[brickIndex]
        [facingIndex])
                        points[vPoint.X + brickPoint.X,
                            vPoint.Y + brickPoint.Y] = (byte)(brickIndex + 1);
                    FreeLine();
                    if (playing && !reviewing) NextBrick();
```

```
        }
        private void ImageListRecreateHandle(object sender, EventArgs e)
        {
            DoChange();
        }
        private void DetachImageList(object sender, EventArgs e)
        {
            ImageList = null;
        }
        public ImageList ImageList
        {
            get
            {
                return imageList;
            }
            set
            {
                if (value != imageList)
                {
                    EventHandler handler = new EventHandler(ImageList
RecreateHandle);
                    EventHandler handler2 = new EventHandler(DetachImage
List);
                    if (imageList != null)
                    {
                        imageList.RecreateHandle -= handler;
                        imageList.Disposed -= handler2;
                    }
                    imageList = value;
                    if (value != null)
                    {
                        brickWidth = ImageList.ImageSize.Width;
                        brickHeight = ImageList.ImageSize.Height;
                        DoChange();
                        if (!playing && !reviewing) GameOver();
                        if (youxiNext != null)
                        {
                            youxiNext.BackColor = BackColor;
                            youxiNext.SetSize(brickWidth * 4, brickHeight *
4);
                            youxiNext.Update(this);
                        }
                        value.RecreateHandle += handler;
                        value.Disposed += handler2;
                    }
                }
```

```
            }
        }
```

6）定义 TetrisScore，设置得到游戏的积分。具体实现代码如下：

```
    private void DetachyouxiScore(object sender, EventArgs e)
        {
            youxiScore = null;
        }
   public youxiScore TetrisScore
        {
            get
            {
                return youxiScore;
            }
            set
            {
                if (value != youxiScore)
                {
                    EventHandler handler = new EventHandler(Detachyouxi
   Score);
                   if (youxiScore != null) youxiScore.Disposed -= handler;
                    youxiScore = value;
                    if (value != null)
                    {
                        value.SetSize(4 * brickWidth, 6 * brickHeight);
                        value.Update(this);
                        value.Disposed += handler;
                    }
                }
            }
        }
```

7）定义 ProgressBar，设置实现回放进度条的处理。具体实现代码如下：

```
    private void DetachProgressBar(object sender, EventArgs e)
        {
            progressBar = null;
        }
        /// <summary>
        /// 回放进度条
        /// </summary>
        public ProgressBar ProgressBar
        {
            get
            {
                return progressBar;
```

```
            }
            set
            {
                if (value != progressBar)
                {
                    EventHandler handler = new EventHandler(DetachProgre
ssBar);
                    if (progressBar != null) progressBar.Disposed -= hand
ler;
                    progressBar = value;
                    if (value != null)
                    {
                        progressBar.Minimum = 0;
                        progressBar.Maximum = StepInfos.Count;
                        progressBar.Value = stepIndex < 0 ? 0 : stepIndex;
                        value.Disposed += handler;
                    }
                }
            }
        }
```

8）定义各个热键事件处理方法，并通过方法 OnKeyDown()设置按键按下时所要发生的对方块的控制方式。例如左移、右移和变换等方式。具体实现代码如下：

```
protected override void OnPaint(PaintEventArgs e)
        {
            base.OnPaint(e);
            if (backBitmap != null) e.Graphics.DrawImage(backBitmap, 0,
0);
        }
        protected override bool IsInputKey(Keys keydata)
        {
            return (keydata == Keys.Down) || (keydata == Keys.Up) ||
                (keydata == Keys.Left) || (keydata == Keys.Right) ||
                (keydata == Keys.Escape) || base.IsInputKey(keydata);
        }
        protected override void OnMouseDown(MouseEventArgs e)
        {
            base.OnMouseDown(e);
            if (CanFocus) Focus();
        }
        protected override void Dispose(bool disposing)
        {
            if (threadReview != null) threadReview.Abort();
            base.Dispose(disposing);
        }
        protected override void OnKeyDown(KeyEventArgs e)
        {
            base.OnKeyDown(e);
```

```
            if (!playing ‖ reviewing) return;
            switch (e.KeyCode)
            {
                case Keys.A: // 左移
                case Keys.Left:
                    BrickOperate(BrickOperates.boMoveLeft);
                    break;
                case Keys.D: // 右移
                case Keys.Right:
                    BrickOperate(BrickOperates.boMoveRight);
                    break;
                case Keys.W: // 变化
                case Keys.Up:
                    BrickOperate(BrickOperates.boTurnLeft);
                    break;
                case Keys.Back: // 变化
                case Keys.F:
                    BrickOperate(BrickOperates.boTurnRight);
                    break;
                case Keys.Down: // 向下
                case Keys.S:
                    BrickOperate(BrickOperates.boMoveDown);
                    break;
                case Keys.Enter: // 直下
                case Keys.Space:
                case Keys.End:
                case Keys.J:
                    BrickOperate(BrickOperates.boMoveBottom);
                    break;
            }
        }
    }
```

9）通过类 youxiNext 来显示下一个出现的方块控件，分别通过方法 youxiNext()、方法 Clear()、方法 SetSize()和方法 OnPaint()来设置下一个方框。

类 youxiNext 的具体实现代码如下：

```
    public class youxiNext : Control
    {
        public youxiNext()
        {
            BackColor = Color.Black;
            base.SetStyle(ControlStyles.OptimizedDoubleBuffer, true);
        }
        private Bitmap backBitmap;
        public void Clear()
        {
```

```
            Graphics vGraphics = Graphics.FromImage(backBitmap);
            vGraphics.FillRectangle(new SolidBrush(BackColor), vGraphics.
ClipBounds);
        }
        public void Update(youxiControl AyouxiControl)
        {
            if (AyouxiControl == null) return;
            Clear();
            AyouxiControl.DrawNext(Graphics.FromImage(backBitmap));
            Invalidate();
        }
        public void SetSize(int AWidth, int AHeight)
        {
            Width = AWidth;
            Height = AHeight;
            backBitmap = new Bitmap(AWidth, AHeight);
        }
        protected override void OnPaint(PaintEventArgs e)
        {
            base.OnPaint(e);
            if (backBitmap != null) e.Graphics.DrawImage(backBitmap, 0, 0);
        }
    }
```

10）通过类 youxiScore 来显示游戏的当前积分，分别通过方法 youxiScore()、方法 Clear()、方法 Update()、方法 SetSize()和方法 OnPaint()来进行积分的处理设置和显示。

类 youxiScore 的具体实现代码如下：

```
   /// 显示积分信息的控件
    public class youxiScore : Control
    {
        private Bitmap backBitmap;
        public youxiScore()
        {
            BackColor = Color.Black;
            base.SetStyle(ControlStyles.OptimizedDoubleBuffer, true);
        }
        public void Clear()
        {
            Graphics vGraphics = Graphics.FromImage(backBitmap);
            vGraphics.FillRectangle(new SolidBrush(BackColor), vGraphics.
ClipBounds);
        }
        public void Update(youxiControl AyouxiControl)
        {
            if (AyouxiControl == null) return;
```

```
            Clear();
            AyouxiControl.DrawScore(Graphics.FromImage(backBitmap));
            Invalidate();
        }
        public void SetSize(int AWidth, int AHeight)
        {
            Width = AWidth;
            Height = AHeight;
            backBitmap = new Bitmap(AWidth, AHeight);
        }
        protected override void OnPaint(PaintEventArgs e)
        {
            base.OnPaint(e);
           if (backBitmap != null) e.Graphics.DrawImage(backBitmap, 0, 0);
        }
    }
```

参 考 文 献

[1] 周长发. C#面向对象编程[M]. 北京：电子工业出版社，2007.
[2] 内格尔，等. C#高级编程（原书第 6 版）[M]. 李铭，译. 北京：清华大学出版社，2008.
[3] 王小科，吕双，等. C#开发典型模块大全–软件工程师典藏[M]. 修订版. 北京：人民邮电出版社，2010.
[4] 国家 863 中部软件孵化器，C#从入门到精通[M]. 北京：人民邮电出版社，2010.
[5] 米凯利斯. C#本质论（原书第 2 版）[M]. 周靖，译. 北京：人民邮电出版社，2009.
[6] 王小科，吕双，梁冰. C#范例完全自学手册[M]. 北京：人民邮电出版社，2009.
[7] 钱哨，李辉剑，李继哲. C#WinForm 实践开发教程[M]. 北京：中国水利水电出版社，2010.

ISBN 978-7-111-36370-5

◎策划编辑：丁 诚 杨 硕

◎封面设计：道乐 dollor 道乐文化 易啸云

《C语言编程新手自学手册》
《Java编程新手自学手册》
《Java Web编程新手自学手册》
《PHP编程新手自学手册》
《Visual C++编程新手自学手册》
《C#编程新手自学手册》
《编程算法新手自学手册》

上架建议：计算机/程序设计

地址：北京市百万庄大街22号
电话服务
社服务中心：(010) 88361066
销 售 一 部：(010) 68326294
销 售 二 部：(010) 88379649
读者购书热线：(010) 88379203

邮政编码：100037
网络服务
门户网：http://www.cmpbook.com
教材网：http://www.cmpedu.com
封面无防伪标为盗版

定价：86.00元(含1DVD)

